RL-TR-97-92, Volume II (of two)
Final Technical Report
September 1997

CERTIFICATION FRAMEWORK VALIDATION FOR REUSABLE ASSETS - CERTIFICATION FIELD TRIAL, VOLUME II (OF TWO)

Data & Analysis Center for Software,
KAMAN Sciences Corporation

Sharon Rohde and Karen Dyson,
of Software Productivity Solutions, Inc.

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Laboratory
Air Force Materiel Command
Rome, New York

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-97-92, Volume II (of two) has been reviewed and is approved for publication.

APPROVED: Deborah A. Cerino

DEBORAH A. CERINO
Project Engineer

FOR THE DIRECTOR: John A. Graniero

JOHN A. GRANIERO, Chief Scientist
Command, Control, & Communications Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL/C3CB, 525 Brooks Road, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 1997	Final Apr 94 - Feb 97

4. TITLE AND SUBTITLE

CERTIFICATION FRAMEWORK VALIDATION FOR REUSABLE ASSETS - CERTIFICATION FIELD TRIAL, VOLUME II (OF TWO)

5. FUNDING NUMBERS

C - F30602-92-C-0158 T/32
PE - 63728F
PR - 2527
TA - 02
WU - 35

6. AUTHOR(S)

Sharon Rohde and Karen Dyson of Software Productivity Solutions, Inc.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Data & Analysis Center for Software
KAMAN Sciences Corporation
Griffiss Business & Technology Park
775 Daedalian Drive
Rome, NY 13440-4909

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Rome Laboratory/C3CB
525 Brooks Road
Rome, NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

RL-TR-97-92, Vol II (of two)

11. SUPPLEMENTARY NOTES

Rome Laboratory Project Engineer: Deborah A. Cerino/C3CB/(31) 330-2054

12a. DISTRIBUTION AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The purpose of this effort was to further develop, apply, and validate the Rome Laboratory Software Certification Framework for designating various levels of confidence in the quality of reusable software. This effort fine-tuned the Framework's ability to distinguish between reusable assets of differing quality.

The effort resulted in a two volume final technical report. Volume I - the Project summary, describes the complete contractual effort. The report discusses how the quality assessment methodology, techniques, and metrics embodied within the Rome Laboratory Software Quality Framework (SQF) could be applicable to the certification of reusable assets. The report discusses potential upgrades and re-engineering the Rome Laboratory Software Quality Framework (SQF). In addition, it also overviews the application of the Certification Framework to a small set of software components (i.e., source code). Volume II - Certification Field Trial, fully details the procedures, collection forms, results, and lessons learned from the application of the certification process to the software components.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Software Certification, Software Assessment and Evaluation	122
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

List of Figures

List of Tables

Contributors to the ATD Project

Listed in alphabetical order, the following persons contributed to the ATD Project:

Lynda L. Burns, Software Productivity Solutions, Inc.

Deborah A. Cerino, Rome Laboratory of the U.S. Air Force Materiel Command

Karen A. Dyson, Software Productivity Solutions, Inc.

Jeffrey A. Heimberger, Software Productivity Solutions, Inc.

Beth Layman, Lockheed Martin Corporation

Holly G. Mills, Software Productivity Solutions, Inc.

Annette Myjak, Software Productivity Solutions, Inc.

Sharon L. Rohde, Software Productivity Solutions, Inc.

Tom Strelich, GRC International, Inc.

Steven Wee, Software Productivity Solutions, Inc.

1 Introduction

This volume of the Final Technical Report (FTR) of the Certification Framework Validation for Reusable Assets describes a certification field trial performed by the prime contractor, Software Productivity Solutions, Inc.

Section 2 of this report details the procedures used to perform the field trial. These procedures are also known as the Certification Framework's Default Certification Process. Information about the derivation of the default process is contained in Volume 2. Section 2 of this report, along with the blank data collection forms in Appendix A, was originally published as a stand-alone document provided to the personnel performing the field trial as an instruction manual.

Section 3 of this report describes the results of the field trial both in terms of the asset certified and the lessons learned by having attempted the field trial. This section includes the completed data collection forms.

Appendix A contains the blank data collection forms used during the field trial.

Appendix B contains the certification defect reports resulting from the certification of the asset in the field trial.

This document, FTR Volume 2 - Additional Certification Field Trial - details the procedures, collection forms, results, and lessons learned from the second certification field trial performed by Software Productivity Solutions, Inc. The following documents serve as supporting information to this document:

- Volume 1 - Project Summary, describes the work performed and the results of the CRC project.
- Volume 2 - Certification Framework (CF) - describes the research conducted to develop the CF.
- Volume 3 - Cost/Benefit Plan - describes a systematic approach to evaluating the costs and benefits of applying certification technology in the context of a reuse program.
- Volume 4 - Operational Concept Document (OCD) - defines the operational concept of an automated certification environment and reports the results of field interviews with potential users.
- Volume 6 - Certification Toolset, identifies the requirements for certification tools and reports the evaluation and selection of tools based on these requirements. Additional supporting information is found in the following succeeding volumes of the project documentation suite:
- Volume 7 - Code Defect Model - provides a model of code defects based on empirical data collected from studies of industry projects.

The details of the work completed in each of these topic areas can be found in the designated supporting document.

2 Field Trial Procedures

This section describes the procedures used in the certification field trial. This section plus Appendix A, Data Collection Forms, was originally published as a stand alone instruction manual for the personnel performing the field trial.

A field trial is an implementation of a technology in a realistic situation under controlled conditions. Field studies permit a more detailed examination of a specific effect than is possible by monitoring normal repository operations. Field studies may be conducted to measure effects of certification other than those captured in cost avoidance models or to obtain a finer calibration of model parameters. Also deficiencies in the data provided by the cooperating repository(s) may be compensated for by field studies.

The purpose of this field trial is to assess the effort required to implement the certification process and its effectiveness in detecting defects in the assets. Sections 2.1 through 2.5 describe in detail the default certification process and the steps necessary to execute each testing procedure. The data collection requirements are outlined in Section 2.7.

2.1 Default Certification Process

The certification field trial will use the default certification process illustrated in Figure 2-1 below.

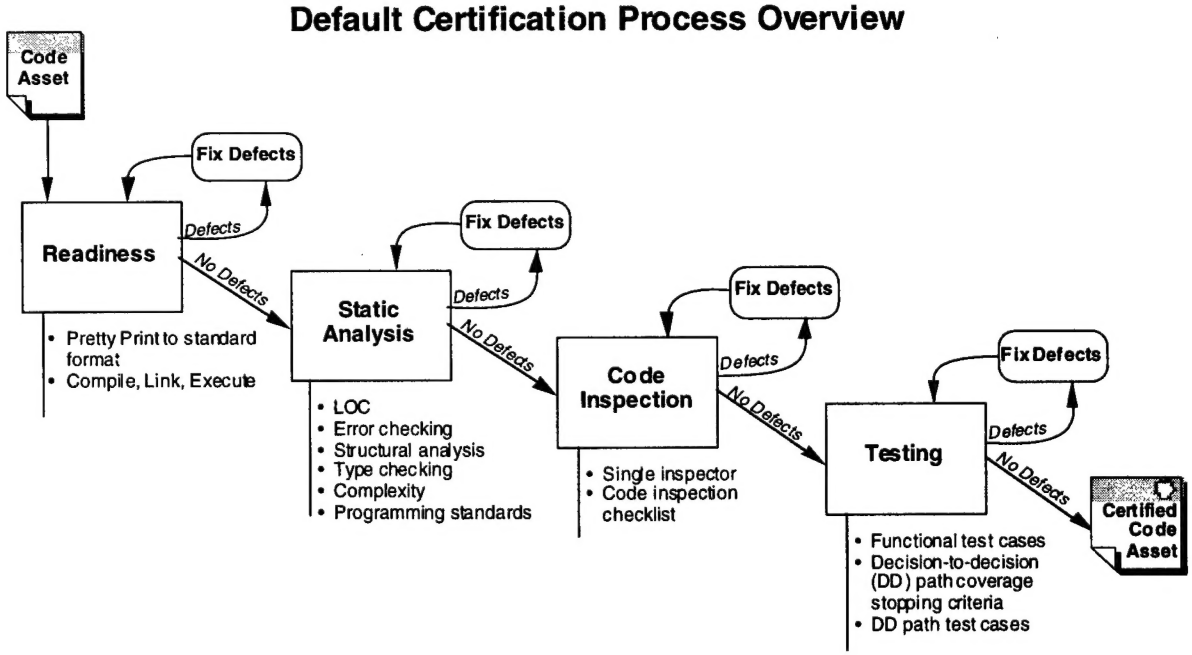

Figure 2-1. Default certification process used in field trial

This default process certifies *code* components (as opposed to other types of reusable assets) and addresses the certification concerns of Completeness, Correctness, and Understandability. The default certification process consists of four main steps which correspond to four increasingly stringent "levels" of certification. Each step of the certification process is discussed in more detail in sections 2.2-2.5.

- **Step 1: Readiness.** The objective of the first step is to demonstrate that the code asset is complete by making sure that it compiles and links successfully, and to prepare it for further certification steps with a pretty printer. This step requires minimal resources.
- **Step 2: Static Analysis.** The second step consists of largely automated static analysis of the code. The process shown in Figure 2-1 lists the analyses to be performed for C++ code. These analyses were selected based on the capabilities of readily available commercial tools. Because this step uses automated static analysis tools, the resources required are mainly for setting up the analysis and interpreting the results.
- **Step 3: Code Inspection.** The third step is an inspection of the code by a single inspector using a reuse certification code inspection checklist. The reuse certification-specific checklist was synthesized from numerous checklists and concentrates on Correctness and Understandability defects. This is a human-intensive technique and requires a software engineer knowledgeable in the implementation language.
- **Step 4: Testing.** The fourth step is a hybrid of functional and structural testing. Functional test cases are constructed. The code is instrumented to record structural coverage information, and all of the functional test cases are executed. If the coverage criterion is met, the testing step is complete; otherwise, the functional test cases are supplemented with structural test cases to achieve the required coverage. Like Step 3, this step is also human-intensive and requires knowledgeable personnel.

The process used in the field trial will include all four steps, so that we can evaluate all of the techniques embodied in the default process. It is not necessary to perform all four steps in practice unless the objective is to certify to the highest level. The certification process could terminate after any step.

Completion Criteria. The steps are intended to be followed in the order shown, and each step must be successfully completed before proceeding on to the next step. No steps are to be skipped. Successful completion means that no major defects are found in that step. If major defects are found, they must be corrected and the step repeated, if necessary, in order to achieve that level of certification. The decision as to whether a step or portion of a step should be repeated depends on the nature of the defect encountered and corrected.

Major defects are defined as defects that

- prevent completion of the current certification step, or

- would result in a failure during testing.

For example, failure to successfully compile would be a major defect. Non-conformance to a style guideline would be a minor defect.

Certification and Quality. Because of the requirement that major defects found must be corrected before a component is considered to have achieved a particular level of certification, we can make certain assumptions about the quality of certified components.

Tool Support Environment. The current tool environment, shown in Figure 2-2, to support this default certification process will be installed on a Dell Pentium PC, 40 MB RAM, running MS-DOS and the Windows 95 environment.

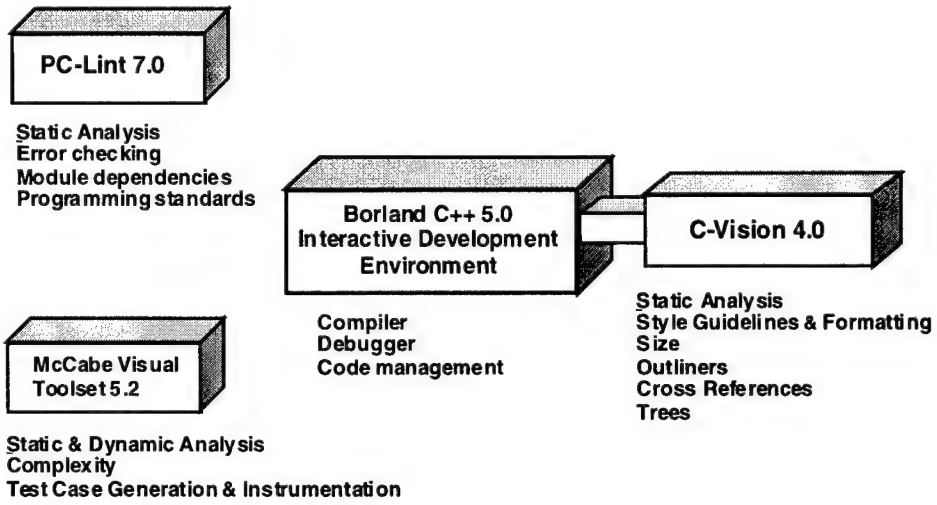

Figure 2-2. Certification Tool Set

The specific tools are the McCabe Visual Toolset, PC Lint and C-Vision. The steps in the certification process provide instructions for when and how these tools should be used and, if necessary, tool substitution guidelines. Tool selection for the C++ Certification Field Trial was made to closely match the functional environment of the Ada Certification Field Trial.

2.2 Procedures for Applying Techniques

The following sections describe in detail the required steps for each of the activities in the default certification process: asset readiness, static analysis, code inspection, and testing. For each activity, the following information is provided:

- Entry Criteria
- Inputs
- Objectives
- Outputs

- Exit Criteria
- Tools
- Procedures

These sections are provided as step by step instructions for executing the default certification process.

Important Note

All defects encountered in performing any step of the process should be documented using the Certification Defect Report found in Section 2.7, Data Collection Plan. Do not record defects for more than one C++ module or separately compilable file on the same report form. If the same type of error is found in multiple places within the same C++ module or separately compilable file, simply note all lines of code in which the error occurs.

2.3 Asset Readiness

Entry Criteria	• Budget: minimal – only requires resources to compile and link the component source code • Personnel skill level: entry level programmer able to operate compiler and construct dummy main program, if needed.
Input	• C++ source code • Source code formatting standards or defaults for pretty printing
Objectives	• Completeness – Demonstration that the component includes all source code comprising the full "include" closure and has no dependencies on missing software. This includes vendor, platform, class libraries and API dependencies. Demonstration that the components can be successfully linked into an executable program.
Output	• Pretty-printed source code • Effort expended on this process • Certification Advancement flag: true if compile and link successful else false • Defect reports, if compile and/or link failure
Exit Criteria	• All steps in the procedure completed. • Definition of success ==> Components compile and link without error • All defects recorded and disposition determined.
Tools	• Borland C++ Compiler/pretty printer • Text editor • Linker
Procedure	• Determine component completeness by compiling component source code to verify complete "include" closure and that the component compiles without error. • If needed, construct dummy main program to "include" (but not call) components. • If appropriate, compile dummy main program and link. • Identify any superfluous code files delivered with the components. • If compilation and linking is successful then pretty-print source code to ensure adherence to source code formatting standards. • Each defect should be reviewed to determine whether it is a major or minor defect. All major defects should be repaired to consider this certification step to be successful, and before proceeding on with the next step in certification. Defects that are not repaired should be reported to reusers.

2.4 Static Analysis

Entry Criteria	• Successful completion of Asset Readiness procedure • Budget: minimal – only requires resources to set-up, execute, and analyze results of automated tools. • Personnel skill level: entry level programmer able to operate compiler and static analysis tools. Must understand basic program structure concepts and semantics.
Input	• Formatted (i.e., pretty-printed) C++ source code • C++ guideline settings (i.e., thresholds, checks enabled/disabled)
Objectives	• Correctness – Identification of computation, logic, data, interface, and other defects. Incorrect control flow and decision structures represent logic defects. Erroneous initialization, definition and accessing data represent data defects. Exception propagation reveals the presence of interface defects and supports the concept of robustness. • Understandability – Demonstration of the degree of compliance with the C++ style and quality guidelines
Output	• Effort expended on this process • Certification Advancement flag: true if no defects or minor defects only; else false. • Defect reports, if any defects are detected
Exit Criteria	• All steps in the procedure completed • All defects recorded and disposition determined
Tools	• C-Vision • PC-Lint • McCabe Visual Toolset
Procedure	• Apply PC-Lint to analyze code structure, control flow and decision logic. • Apply PC-Lint to detect erroneous initialization definitions and data access (data defects). • Apply McCabe Visual Toolset to determine thresholds of cyclomatic complexity, design complexity, and integration complexity (logic defects). • Apply PC-Lint to check for errors across modules (interface defects) • Apply PC-Lint to check compliance with C++ guidelines (computational, logic, data, interface, and other defects) • Each defect should be reviewed to determine whether it is a major or minor defect. All major defects should be repaired to consider this certification step to be successful, and before proceeding on with the next step in certification. Defects that are not repaired should be reported to reusers.

2.5 Code Inspection

Entry Criteria
- Successful completion of Static Analysis procedure
- Budget resources—one person tool
- Code has successfully compiled and linked with no errors

Inputs
- Code processed by pretty printer
- Functional description of component
- Code inspection checklist, for applicable language

Objectives
- Correctness – evaluation according to the inspection checklist
- Understandability – evaluation according to the inspection checklist
- Completeness – assessment of the adequacy of the functional description

Outputs
- Defect reports, if any defects are detected (note which checklist item or inspection activity prompted isolation of the defect)
- Effort expended on this process
- Subjective evaluation of checklist items for understandability, objectivity, organization, etc.
- Observations: undocumented features, items for test, portability concerns, copyrights, design issues
- Certification Advancement flag: true if no defects or minor defects only; else false

Exit Criteria
- All code statements inspected
- All checklist items answered (Yes, No, or Not Applicable)
- All defects recorded and disposition determined
- Definition of success ==> no major defects found, or all major defects corrected

Tools
- C-Vision, for:
  - code outlining
  - cross references
  - call tree

Procedure The code inspection procedure is to be applied by a single inspector. The inspector should record effort expended separately for each of the three steps of the procedure. The purpose of the code inspection is to assess the implementation, rather than the design, of the component. The assumption is that the design of the component, as expressed by the functional description, is correct. If the inspection reveals doubts about the design, such information should be recorded in the Observations.

1. Preparation

The purpose of the preparation step is for the inspector to familiarize himself with these aspects of the component listed below. There are no specific outputs of this step.

- **component's functional description** The functional description may be a separate document, or it may simply be the prologue of comments and the description of the component's interface from the C++ header.
- **overall structure of the component** For example, how many modules comprise the component, how many functions are in the packages, and how the modules are related in terms of the calling structure. The purpose of analyzing the overall structure is to give the inspector an idea of the magnitude of the inspection task, i.e., how many items. This information may be obtained by generating a call tree diagram using the McCabe Visual Toolset.
- **review results of static analysis** All major defects found during static analysis should have been corrected prior to starting code inspection. A review of the defects found during static analysis, both major and minor, may provide the inspector with insight into what aspects of the code should receive special attention during inspection.

2. Inspection and Recording of Defects

During this step, the inspector assesses each item on the inspection checklist, in order, one at a time. If you spot a defect associated with a checklist item that you haven't gotten to yet, make a brief note and move on. As each checklist item is completed, it must be marked as Yes, No or Not Applicable.

When a defect is found, it should be immediately be recorded on a defect report form. It is important to note the exact line(s) of code associated with the defect, and which inspection checklist item lead to its discovery. Some inspectors also like to annotate a hard copy of the code listing to avoid inadvertently recording the same defect twice. It is also important to classify the defect by type. All checklist items are preclassified to make this easier.

3. Disposition of Defects

The purpose of this step is to determine whether or not to correct the defects found during code inspection, and to perform the required repair activity. Each defect should be reviewed to determine whether it is a major or minor defect. All major defects should be repaired to consider this certification step to be successful, and before proceeding on with the next step in certification. Defects that are not repaired should be reported to reusers.

To assist in the Code Inspection step, we used the five classes of defects defined in the CRC Volume 7 - Code Defect Model:

Computational: Any defect of a computational or mathematical nature

Logic: Any defect in any logical construct of the code or algorithm, including defects in control flow and decision structures

Data: Any defect related to the usage, initialization, definition, or access of any data defined by or used in the code

Interface: Any defect in how the code uses or interacts with any internal code objects or any external objects, such as the operating system, files, hardware devices, and other software components

Other: Any other defect that does not fit one of the previous categories, such as defects in documentation, programming standards, or unclassified defects.

Checklist Development. For the second Field Trial, formatting conventions were observed to document updates and refinements to the code checklist as an attempt to preserve the integrity of the checklist from the first Field Trail. Specifically, italicized checklist questions indicate those questions used for Ada source code in the first Field Trial. Non-italic questions are those that have been added to modify the existing Ada checklist for a C++ source code component in the second Field Trial. Strikethroughs in the checklist questions indicate that the question was valid for an Ada source code component, but was not appropriate for a C++ source code component due to specific language characteristics. The following references were used to generate the C++ checklist: [BAL92], [DST96], [FAG96], [FAU94], [GER95], [HUM95], [KOE92], [KOE95], [MCC96], [POT94], [SOF95], [SOF96], and [VAN95]. Additional details about the development of the checklist are found in ATD Volume 1 - Project Summary.

Checklist identifiers. Each checklist item has an alphanumeric identifier. The first letter indicates the defect type, and the last letter indicates whether it is an Understandability (U) or Correctness (C) defect. Correctness defects are typically classified as major while Understandability defects are typically classified as minor.

Reuse Certification Code Inspection Checklist for C++

Identifier	Question	Answer
	• **Computational** •	
C.01.U	*For functions that perform computations, are accuracy tolerances documented?* For functions that perform computations, are accuracy tolerances documented for variable types that hold data?	Yes / No / NA

Identifier	Question	Answer
C.02.C	*Do all computations use variables with consistent types, modes, and lengths (e.g., no boolean variables in arithmetic expressions, or mixed integer and floating-point)?* Do all computations use variables with consistent types and/or type casting, values, and lengths? (i.e., no boolean variables in arithmetic expressions) If variable types are mixed, are expected outcomes anticipated and external to the program block?	Yes / No / NA
C.03.C	*Are all expressions free from the possibility of an underflow or overflow exception?*	Yes / No / NA
C.04.C	*Are all expressions free from the possibility of a division by zero?*	Yes / No / NA
C.05.C	*Is the order of computation and precedence of operators correct in all expressions?*	Yes / No / NA
C.06.C	*Are all expressions free from invalid uses of integer arithmetic, particularly divisions?*	Yes / No / NA
C.07.C	*Are all computations free from non-arithmetic variables?*	Yes / No / NA
C.08.C	*Are all comparisons between variables of compatible data types, modes, and lengths?* Are all comparisons between variables of compatible data types, type cast data types, and lengths?	Yes / No / NA
C.09.C	Do all comparisons avoid equality comparison of floating-point variables?	Yes / No / NA
C.10.C	Is the code free from assignment of a real expression to an integer variable?	Yes / No / NA
C.11.C	*Are all bit manipulations correct?*	Yes / No / NA
C.12.C	Is the "%" modulus operator used correctly (i.e. not intended as a percentage)?	Yes / No / NA
C.13.C	Is the "/" division operator used to accommodate a discarded remainder?	Yes / No / NA
C.14.C	Are compound operators assigned correctly?	Yes / No / NA
	• Data •	
D.01.C	*Are all data items referenced?*	Yes / No / NA

Identifier	Question	Answer
D.02.U	*Do all references to the same data use single unique names?*	Yes / No / NA
D.03.C	*Are all character strings complete and correct, including delimiters?* Are all character strings and character arrays complete and correct, including delimiters (i.e., value is assigned and enough elements are reserved to hold entire character string and terminating null zero)?	Yes / No / NA
D.04.C	*Are illegal input values systematically handled?*	Yes / No / NA
D.05.C	*Are all variables set or initialized before referenced?*	Yes / No / NA
D.06.C	*Are all array indexes integers?*	Yes / No / NA
D.07.C	*For all references through pointer variables, is the referenced storage currently allocated?*	Yes / No / NA
~~D.08.C~~	~~*Are all storage areas free from alias names with different pointer variables?*~~	~~Yes / No / NA~~
D.09.C	*Are all variables correctly initialized?* Are all variable and constants correctly initialized?	Yes / No / NA
D.10.C	*Are all variables assigned to the correct length, type, storage class and range?* Are all variables and constants assigned to the correct length, type, sign, precision, and range?	Yes / No / NA
D.11.U	*Is the code free from variables with similar names (e.g., VOLT and VOLTS)?* Is the code free from variables and constants with similar names (e.g., VOLT and VOLTS)?	Yes / No / NA
D.12.C	*Are all indexes properly initialized?* Are all indexes properly initialized (i.e., start at zero)?	Yes / No / NA
D.13.U	*Are all data declarations commented?*	Yes / No / NA
D.14.U	*Are all data names descriptive enough?*	Yes / No / NA
D.15.C	*Are constant values declared as constants and not as variables?* Are constant values used as numbers, characters, words, or phrases?	Yes / No / NA

Identifier	Question	Answer
D.16.C	For all arrays or enumeration types, are ranges used for each data type instead of numeric literals?	Yes / No / NA
D.17.U	*Are error tolerances documented for all external input data?*	Yes / No / NA
D.18.U	Are variable names in lower case as is the customary convention?	Yes / No / NA
D.19.U	For object-oriented code, are the first letters of class names capitalized as is the customary convention?	Yes / No / NA
D.20.U	Are upper case letters used for "#define" directives as is the customary convention?	Yes / No / NA
D.21.U	Are "#define" statement used judiciously?	Yes / No / NA
D.22.C	Are assignment equals "=" and equals to "==" operators used correctly?	Yes / No / NA
D.23.C	Have assignment expressions been included in the same condition as the logical test?	Yes / No / NA
D.24.U	Are parenthesis used in the expressions of the "sizeof" operator (i.e., in "sizeof data", parentheses is optional, but it is good programming to include (); Are parenthesis used in the expressions of the "sizeof (data type) where parentheses are required?	Yes / No / NA
D.25.C	Are bitwise operators, bitwise shift, and compound bitwise shift used correctly (i.e., &, vertical bar, ^, ~, >>, <<, <<=, >>=)?	Yes / No / NA
D.26.C	For object-oriented components, do classes have any virtual functions? If so, is the destructor non-virtual?	Yes / No / NA
D.27.C	For object-oriented components, do classes have all three necessary copy-constructors, assignment operators, and destructors?	Yes / No / NA
D.28.C	For object-oriented components, do all structures and classes use the "." reference?	Yes / No / NA
D.29.C	Are all pointers initialized to "null", deleted only after "new", and new pointers deleted after use?	Yes / No / NA
D.30.C	Are names used within the declared scope?	Yes / No / NA

Identifier	Question	Answer
D.31.C	For object-oriented components, is each class declared and implemented in a single file (i.e., with the exception of helper classes packaged with the primary file)?	Yes / No / NA
D.32.C	Are function arguments free from variable argument lists (...) to avoid the inherently type-unsafe?	Yes / No / NA
D.33.U	Is multiple inheritance avoided?	Yes / No / NA
D.34.U	Are "return" types always provided, even if "void"?	Yes / No / NA
D.35.C	For object-oriented components, does every constructor initialize every data member in its class?	Yes / No / NA
D.36.C	For object-oriented components, do assignment operators correctly handle assigning an object to itself?	Yes / No / NA
D.37.C	Is "delete [ ]" used when deleting an array to determine the size of the array being deleted?	Yes / No / NA
D.38.U	For object-oriented components, are object fine grained?	Yes / No / NA
D.39.U	For object-oriented components, is the object encapsulated (i.e., highly related methods and data isolated)?	Yes / No / NA
D.40.U	For object-oriented components, is there low dependency between objects?	Yes / No / NA
D.41.U	For object-oriented components, do objects exhibit high fan in?	Yes / No / NA
	• Interface •	
~~I.01.C~~	~~*Are all propagated exceptions declared as visible and documented?*~~	~~Yes / No / NA~~
~~I.02.C~~	~~*Are all propagated exceptions handled (not raised) by the calling unit?*~~	~~Yes / No / NA~~
I.03.C	*Are reasonable ranges declared for all output values?*	Yes / No / NA
I.04.C	*For all global variables, is their use justified, and are they documented?*	Yes / No / NA
I.05.U	*Are all subprogram parameter modes shown and usage described via comments?* Are all subprogram parameter types shown and usage described via comments?	Yes / No / NA

Identifier	Question	Answer
I.06.U	*Does the prologue document all side effects, such as propagated exceptions?* Does the prologue document all side effects?	Yes / No / NA
I.07.U	*Are the interface data items free from negative qualification logic (e.g., boolean values that return "true" upon failure rather than success)?*	Yes / No / NA
I.08.C	*Do all units systems of formal parameters match actual parameters (such as degrees vs. radians, or miles per hour vs. feet per second)?*	Yes / No / NA
I.09.C	Are all functions free from modification of input parameters?	Yes / No / NA
I.10.C	*Are global variables consistently used in all references?*	Yes / No / NA
I.11.C	*Are files opened before use and closed when finished?* Are files opened immediately prior to access and closed as soon as done?	Yes / No / NA
I.12.C	*Are all input parameter variables referenced? Are all output values assigned?*	Yes / No / NA
I.13.U	*Does each unit have a single function, and is it clearly described?*	Yes / No / NA
I.14.C	*Are all functions free from side effects?*	Yes / No / NA
I.15.C	*Is there a single entry and a single exit?*	Yes / No / NA
I.16.C	Does the program and all its functions end with a return statement?	Yes / No / NA
I.17.C	Does each return have a closing brace (i.e., after the end of a block, the end of the main function [main ( )], and the end of the program?	Yes / No / NA
I.18.C	Are the widths and formats of numbers specified correctly for printing?	Yes / No / NA
I.19.C	Are the most frequently executed statements in a "switch" arranged at the top of the list to improve the efficiency of the code?	Yes / No / NA
I.20.C	If "ios::out" is used to open a file for writing (i.e., C++ creates the file), does it overwrite the filename that exists?	Yes / No / NA

Identifier	Question	Answer
I.21.U	Is code free from "non-standard" syntactic constructs such as unconventional preprocessor directives?	Yes / No / NA
I.22.C	Is passing objects by value, or by reference avoided (e.g., where implicit conversions result in member wise copying)? Are dynamically allocated application objects passed as pointers?	Yes / No / NA
I.23.C	To decrease performance overhead, are local variables created and assigned at once?	Yes / No / NA
I.24.C	Are files properly declared, opened, and closed?	Yes / No / NA
I.25.C	Is a file closed in the case of an error return?	Yes / No / NA
I.26.C	Are all "include" statements complete?	Yes / No / NA
I.27.C	Are "inline" functions used only when performance is needed?	Yes / No / NA
I.28.C	Are "new" and "delete" used to allocate and deallocate storage rather then "malloc" and "free" (i.e., which are type-unsafe)?	Yes / No / NA
I.29.C	Have timing, sizing, and throughput been addressed?	Yes / No / NA
	• **Logic** •	
L.01.C	*Are all negative boolean and compound boolean expressions correct?*	Yes / No / NA
L.02.C	*For all case statements, is the domain partitioned exclusively and exhaustively?* For all "switch" statements, is the domain partitioned exclusively and exhaustively?	Yes / No / NA
L.03.C	*Are all indexing operations and subscript references free from off-by-one defects?*	Yes / No / NA
L.04.C	*Are all comparison operators correct?*	Yes / No / NA
L.05.C	*Are all boolean expressions correct?*	Yes / No / NA
L.06.C	*Is the precedence or evaluation order of boolean expressions correct?*	Yes / No / NA
L.07.C	Do the operands of boolean expressions have logical values (0 or 1) or a non zero value which is interpreted as true?	Yes / No / NA

Identifier	Question	Answer
L.08.C	*Does every loop eventually terminate?*	Yes / No / NA
L.09.C	*Is the program free from goto statements?* Are "gotos" used judiciously or can other code be substituted?	
L.10.C	*Are all loops free from off-by-one defects (i.e., more than one or fewer than one iteration)?*	Yes / No / NA
L.11.C	Are all switch statements free from "others" branches?	Yes / No / NA
L.12.C	*Are all decisions exhaustive?*	Yes / No / NA
L.13.C	*Are end-of-file conditions detected and handled correctly?*	Yes / No / NA
L.14.C	*Are end-of-line conditions detected and handled correctly?*	Yes / No / NA
L.15.C	*Do processes occur in the correct sequence?*	Yes / No / NA
L.16.C	*Are all loops free from unnecessary statements?*	Yes / No / NA
L.17.C	*Are all loop limits correct?*	Yes / No / NA
L.18.C	*Are all branch conditions correct?*	Yes / No / NA
L.19.C	Are loop index variables used only within the loop?	Yes / No / NA
L.20.C	Are all loops free from loop index modification?	Yes / No / NA
L.21.C	*Is all loop nesting in the correct order?*	Yes / No / NA
L.22.U	Do all loops have single exit and entry points?	Yes / No / NA
L.23.U	For all nested loops, are loops and loop exits labeled?	Yes / No / NA
L.24.C	Is the ternary conditional operator "?:" used correctly?	Yes / No / NA
L.25.C	Are the increment and decrement operators properly used in postfix and prefix order?	Yes / No / NA
L.26.U	Do braces surround the body of a "for" and "while" loop even though it only has one statement (i.e., exhibiting good programming practices)?	Yes / No / NA
L.27.U	Are the expected executions anticipated with "while", "do while", and "if while", even though the code will compile?	Yes / No / NA
L.28.C	Are "exit (status)", "break in case", and "break and continue" used to correctly exit the program or exit the loop?	Yes / No / NA

Identifier	Question	Answer
L.29.C	Are counters initialized to zero and the increment operator (i.e., "++") used appropriately?	Yes / No / NA
L.30.C	When "for" loops are used, is the intent for the condition to be tested at the top of the loop (i.e., is the condition ever "True" so that the loop executes)?	Yes / No / NA
L.31.C	Is redundancy eliminated in "for" loops for better efficiency?	Yes / No / NA
L.32.C	Do all "switch" statements contain a default branch to handle unexpected cases?	Yes / No / NA
L.33.C	Does logic handle bad input as well as good input?	Yes / No / NA
	• Other •	
O.01.U	*Is the descriptive prologue complete and correct?*	Yes / No / NA
O.02.C	*Are all printed or displayed messages free from grammatical or spelling errors?*	Yes / No / NA
O.03.U	*Does the code follow basic structured programming techniques?*	Yes / No / NA
O.04.U	*Are all assumptions documented?*	Yes / No / NA
O.05.C	*Is the code written only in Ada?* Is the code written only in C or C++?	Yes / No / NA
O.06.U	Is each variable declared on a single line to improve readability and maintainability?	Yes / No / NA
O.07.U	Does code contain mapping to parent documents, or functional specifications?	Yes / No / NA

2.6 Hybrid Structural-Functional Testing

Entry Criteria	• Code Inspection procedure has been successfully completed • Code is complete with respect to executability • Functional specification or description is available
Inputs	• Source code and pretty printer listing • Functional specification or description • Test requirements derived during code review (if any)
Objectives	• Correctness – determination of whether the component correctly performs its intended function within the specified requirements. • Completeness – determination of whether the component is complete with respect to the functional specification or description. • Understandability – assessment of the understandability of the functional specification or description.
Outputs	• Test cases • Coverage metrics • Result summary report • Effort expended on this process • Defect reports, if any defects are detected • Certification Advancement flag; true if no major defects found, else false
Exit Criteria	• All functional tests completed • At least 90% decision-to-decision (DD) path coverage for 100% of the components has been achieved • All defects recorded and disposition determined
Tools	• McCabe Visual Toolset for unit level test case generation and unit level instrumentation.

Structural-Functional Testing Procedure

The basic procedure is to develop functional test cases, and to instrument the code to measure logical branch coverage, also known as decision-to-decision (DD) path coverage. Run the functional test cases, and if at least 90% DD path coverage has been achieved on 100% of the components (with the possible exceptions noted below), testing is complete. If the coverage criterion has not been met, then supplement the functional test cases with structural test cases until the coverage criterion has been met.

1. Instrument code for DD path coverage using the McCabe Visual Toolset.

2. Design functional test cases by following the **Functional Test Case Creation Guidelines** listed below.
3. Create test harness (driver and stubs) if required to run the functional test cases.
4. Run each functional test case. Compare the actual test outputs to the expected outputs, determine if there is a difference and, if so, a defect in the code has been detected.
5. Fill out a Defect Report for each defect detected.
6. Determine DD path coverage achieved by the complete suite of test cases by running the McCabe Visual Toolset (see Appendix B for detailed instructions). If all components have at least 90% coverage, then testing is complete—go on to the next step. Otherwise, create supplemental structural test cases to increase the coverage. Use the McCabe Visual Toolset to display DD paths not executed, and determine the values the decision variables must have in order to execute these paths. Repeat steps 2-6 for the supplemental test cases until the coverage criterion is met.
7. Prepare test summary report, summarizing the test cases, noting the defects (if any) that were found, and stating the coverage achieved.
8. Review each defect to determine whether it is a major or minor defect. All major defects should be repaired to consider this certification step to be successful. Defects that are not repaired should be reported to reusers.

Exceptions to Coverage Requirements. The test coverage requirement of 90% DD path coverage is a key to achieving certification. The exceptions listed below describe cases that may prevent achievement of DD path coverage. The most efficient approach to certification is probably to achieve the best coverage possible with these exceptions, and then to make the necessary changes to the asset and continue testing. Another possibility is to certify the asset subject to exceptions which are then well documented in the certification results.

For cases a and b, the defective code must be repaired before certification can continue. For case c, the decision about certification depends on whether the entire reused module is intended to be made available as a separate entity. If so, then additional test drivers may be needed to exercise it completely. For case d, it may be difficult to trigger an exception to execute the path.

a. Paths blocked by defects. Defects must be repaired before coverage can be increased and certification can be achieved.
b. Leftover debugging code, or dead code. This code should be removed to verify that it is superfluous so that certification can be achieved.
c. Code which is intentionally never executed, such as unused functions in reused packages that are used by the asset being certified.
d. Non-specific "others" exception handlers.

Functional Test Case Creation Guidelines

1. Using the functional specification for the code and/or documentation within the code, identify the functional requirements of the code (what it is intended to do and any restrictions, limitations, or special conditions on how it performs its function(s)), the input and output variables, and the allowable ranges of values for the input and output variables. Determine (on a functional, not code structure, level) how the input variables are combined and processed to produce the desired outputs.

2. Create at least two equivalence classes for each input variable that at a minimum divide the possible input values into valid and invalid values. Equivalence classes partition the possible input values into disjoint sets (classes) such that any input value in one class should result in an output equivalent to that resulting from any other input value selected from that same class. Create equivalence classes for variables that are defined by bounded or discrete ranges according to the following rules:

   - If the allowable values for the input variable are defined as a range between two values, create three equivalence classes - values below the lower limit, values within the specified range, and values above the upper limit. For example, if an input velocity was specified as $0 \leq$ velocity ≤ 100 mph, the three classes would be velocity < 0, $0 \leq$ velocity ≤ 100, and velocity > 100.

   - If the allowable values for the input variable are defined as a discrete range, create equivalence classes for each of the values that are treated differently from other variables. For example, if an input vehicle variable was specified to be either a bicycle, car, truck, boat, or plane, and if the code processes the input differently according to type of vehicle, then there are six equivalence classes: vehicle = bicycle, vehicle = car, vehicle = truck, vehicle = boat, vehicle = plane, and vehicle $\neq$ bicycle, car, truck, boat, or plane. However, if the code processes the input differently according to the physical environment in which the vehicle operates, then there are four possible equivalence classes: vehicle = bicycle, car, or truck; vehicle = boat; vehicle = plane; and vehicle $\neq$ bicycle, car, truck, boat, or plane.

   Pick at least one input value from each equivalence class, making sure to include a value at the boundary of each class.

3. Identify pseudo-boundary conditions and, for each condition, pick at least one input value that would cause it to arise. A pseudo-boundary condition is a combination or use of variables in the code that makes invalid some otherwise valid input value for the variables. For example, consider a code component ratio_a whose function is to compute the function z=(x/y)*c, where x and y are inputs of measures of some physical phenomena whose values are expected to fall within specified numerical ranges and c is a constant. In this example, the use of the variable y as a divisor is a pseudo-boundary condition: since a zero-divide condition could result, y = 0 is an

invalid input. The code should be tested with y = 0 to determine if zero-divide conditions would arise and be handled properly.

4. Identify equivalence classes for each output variable following the procedure described for input variables. Determine the input values required to produce each class of outputs. For the ratio_a example, creating these equivalence classes addresses whether or not combinations of input values for x and y could result in out of range values for z.

5. Create test cases by combining the inputs determined in steps 3 - 4 so that each input is tested without unnecessary duplication. For example, an input for a pseudo-boundary condition might also be an input for one of the input equivalence classes or one of the output equivalence classes.

6. Using the functional requirements of the code, predict the expected outcome of each test case. An outcome is a change (or the absence of a change) in anything observable as a result of executing a test, including changes in memory, mass storage, I/O devices, registers, and output variables[1]. For the ratio_a example, an expected outcome is an output value for z for each pair of input values x and y and can be determined by a simple computation. For a stack management routine, the contents of a stack as well as a returned entry, a return code, or a pointer are possible outcomes. In cases where the expected outcomes can not be easily computed or derived from the inputs, an oracle or best engineering judgment can be used to predict them.

---

[1] Beizer, Boris. *Software Testing Techniques.* 2nd Ed. New York: Van Nostrand Reinhold, 1990.

2.7 Data Collection Plan

The objective of this field trial is to evaluate the overall effectiveness of the certification process in assessing the correctness of the asset. This can be used as an indicator of avoidance within a reuse context. In order to evaluate effectiveness, data must be collected in four basic categories in the field trial: (1) product characteristics, (2) certifier profile, (3) process execution, and (4) certification defect profile.

Product Characteristics

Descriptive information about the assets being certified needs to be collected in order to assess its impact on the effort required to execute the certification process. The following information should be recorded for each asset:

Table 2-1. Product Characteristic Data Elements

ASSET NAME
Origin of Asset
Application Domain
Purpose of asset
Language
Number distinct "packages" contained in the asset
Physical lines of code (non-blank lines)
Supporting packages
Age of asset
Version number of asset
Previous inspection and testing activities
Additional documentation

Certifier Profile

In addition, the knowledge and experience of the certifier has an impact on the effectiveness of performing the certification procedures. This data must be collected in order to measure the impact of certifier's skills. It is assumed that one person will perform all of the procedures; however, if more than one person is involved in the process, a form should be filled out for each certifier. Also, this must be noted in the Certifier Identification block provided in the process implementation data sheets (in Appendix A).

Table 2-2. Certifier Profile Data Elements

CERTIFIER IDENTIFICATION
Number of years of programming experience
Number of years of programming experience in asset's language
Education (list degrees)
Tool experience (hours with each tool) **(note: before starting certification process)**

Process Execution

The two major components of evaluating process execution are (1) the level of effort required to complete each procedure and the number of defects found during each activity. This information, in conjunction with the other data categories, will permit a very general "cost-effectiveness" assessment of this overall certification process. See Appendix A for the specific data collection forms.

Table 2-3. Process Characteristic Data Elements

	Default Certification Activities			
	Readiness	**Static Analysis**	**Code Inspection**	**Testing**
Certifier Name or ID Number				
Level of Effort (hours)				
Number of Defects Found				
Computational				
Data				
Interface				
Logic				
Other				
Total				
Problems in Applying Techniques				
Problems in Using Tools				
Problems with Process Guidance				

Certification Defect Report

It is important to record the defects detected at each step in the process. Defects found should be classified according to type (computational, data, interface, logic, other).

It is not intended that the field trial include repairing defects unless the defect impairs further certification steps. If defects are repaired, the effort to repair should be recorded. If the defect is not isolated by virtue of the certification technique (e.g., a test case results in a failure that must then be traced to a line of code), then the effort to isolate should be recorded separately from the effort to repair.

A standard Defect Report form is shown below. All data collection forms are provided in Appendix A for convenient data collection.

◆ Certification Defect Report ◆

Defect Report Identifier _____

Severity ☐ Major ☐ Minor

Asset Identifier _____

Originator _____

Defect Type
- ☐ Computational
- ☐ Data
- ☐ Interface
- ☐ Logic
- ☐ Other _____

Technique Used
- ☐ Readiness
- ☐ Static Analysis
  - ☐ Data Flow
  - ☐ Order Dependency
  - ☐ Alias Usage
  - ☐ Unreachable Code
  - ☐ Style Guideline
  - ☐ Other _____
- ☐ Code Inspection ☐ Item _____
- ☐ Testing
  - ☐ Functional Test Case
  - ☐ Structural Test Case
  - ☐ Other _____

Tool Used
- ☐ C-Vision
- ☐ PC-Lint
- ☐ McCabe Toolset
- ☐ C++ env. (compiler, etc.)
- ☐ None

Code Line Number(s) _____

Effort to Isolate _____

Effort to Repair _____

Description of Defect:

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

3 Results

This section presents the results of the second field trial performed by SPS. The first subsection is an overview of the field trial. The next subsection presents the results of the certification of the asset in terms of defects found. The final subsection discusses the lessons learned as a result of the second field trial. The first field trial is documented in CRC Volume 5.

3.1 Field Trial Overview

The certification field trial described in this report was performed by SPS personnel Sharon Rohde, Pat Aymond and Karen Dyson.

Personnel. Ms. Rohde was selected to perform the field trial because of her experience with the C++ language, and also because she was not involved in the derivation of the default certification process or in writing the field trial procedures (in Section 2). Ms. Rohde installed the certification tools and performed all of the certification steps.

Ms. Dyson was a contributor to the derivation of the default certification process and co-author of the field trial procedures. Pat Aymond selected the asset to certify and seeded additional defects into the asset, consulting with Karen Dyson. Ms. Dyson served as consultant for the analysis of the results and Lessons Learned.

Objectives

The objectives of the field trial were as follows:

- Perform all of the steps in the default certification process
- Use all of the tools in the certification tool set
- Assess the accuracy and understandability of the procedures guidance
- Collect effort and technique effectiveness data
- Select a single asset to certify sized for a 2 staff-week certification effort

While technique effectiveness data was collected, the field trial was not intended to be an experiment to determine the effectiveness of the techniques that comprise the default certification process. The design and implementation of an experiment of that type is quite involved and is significantly beyond the scope of the CRC/ATD contract. The effort and technique effectiveness data was collected in order to compare the actual results with comparable values culled from other research studies.

Accomplishments

All of the above objectives were satisfied by the field trial.

3.2 Asset Certified

The resources allocated to the field trial task allowed for certification of a single asset. The asset to certify was selected based on its similarity with the asset certified in the Ada field trial. Since the default certification process was derived for Ada code assets, it was modified for a C++ code asset. The Reuse Code Inspection Checklist was modified for a C++ code asset.

Size. It was estimated that an asset of about 1000 logical lines of code would be large enough to not be trivial and yet small enough to be certified in a 2 staff-week effort. The effort constraint was developed based on extensive interviews of reuse library personnel performed early in the CRC contract [see the CRC Volume 4 - Operational Concept Document], which indicated that 2 staff-weeks were about the right amount to devote to certifying a single asset.

Defect history. In order to assess the effectiveness of the certification process at finding defects, it was necessary to have an asset with defects known in advance. To achieve this requirement of a defect history, we seeded defects into the selected component to the similar extent as the Ada component in the previous initial field trial.

Selected Asset

The selected asset was a labels program packaged with the Borland compiler as example code. This single executable program automatically generates mailing labels from a master list. It reads a subscription list, inserts new subscriptions into a master list, and prints the contents of the master list in a standard label format. It had no recorded defect history.

Size of Asset

Lines of code (physical)	3,356 lines (est.)
Number of class libraries	3
Number of supporting "C"files	2

The size of the asset was determined by counting lines which included compiler directives and the following header files with seeded defects:

<classlib\listimp.h>

<classlib\objstrm.h>

<classlib\date.h>

An informal desk check type code review turned up no major or minor defects. Therefore we decided to seed 14 additional major defects in order to have a significant number of major defects known in advance of the field trial. The seeded defects are summarized in the table below. All seeded defects are documented in Appendix B and have an identifier starting with "PA_". These known defects were not shown to Ms. Rohde prior to or during the field trial.

Summary of Seeded Defects

Identifier	Unit	Lines	Description	Type
PA_001	labels.cpp	386-387	Changed write to read output file in subscription list destructor. Does not write subscriptions to master file.	Interface
PA_002	date.h	31	Changed value of constant Julian date of 1/1/1901 from "2415386L" to "1415386L"	Data
PA_003	date.h	106	Changed operator "-=" to "=" and data type from integer to constant.	Interface
PA_004	date.h	253,256	Changed operator "-=" to "=" in inline operator definition.	Interface
PA_005	date.h	272	Changed inline function that checks for valid months so that months January and December are not valid.	Logic
PA_006	listimp.h	82,83,91	Instead of zeroing out the list element counter, it was set to 1.	Logic
PA_007	listimp.h	719	In ForEach function, incorrect while condition does not iterate through list properly.	Logic
PA_008	listimp.h	889	Changed notation from class name to arithmetic operator.	Logic
PA_009	objstrm.h	299	Address of object is not stored in database.	Logic
PA_010	objstrm.h	626	Changed inline function clear, changed "hardfail" to "basefield".	Data
PA_011	objstrm.h	1020	Improper terminator in switch statement; changed "break" to "switch".	Logic
PA_012	labels.cpp	414	Wrong while loop condition, changed "iter !=0 to "iter == 7". Will not correctly write susbscription list to output file.	Logic
PA_013	labels.cpp	429	Incorrect initialization of for loop iterator; changed "i = 0" to "i = 11". Will not read in subscriptions from master file unless count > 11.	Logic
PA_014	labels.cpp	605	Changed type declaration of main routine from "int" to "unsigned int".	Interface

The seeded defects were not created in an attempt to duplicate a particular defect profile (i.e., distribution of defect types). There are more logic defects than other types simply because these are the easiest type to invent. It turned out to be rather more difficult than we anticipated to create defects that were not caught by the compiler, nor caused immediate catastrophic failure on execution.

In the results section below, we look at all of the known defects in the certified asset after having completed the certification process.

3.3 Certification Results

This subsection presents the results of the second certification field trial performed by SPS. Analysis of the data collected during the field trial and of the defects found in the asset are included in these results. Lessons learned are discussed in the next subsection.

Data collection forms described in Section 2.7 were completed during the field trial. All certification defect reports are in Appendix B, and the other completed forms are contained in this subsection under the appropriate topic.

Staff Experience

As mentioned the overview in subsection 3.1, three SPS personnel were involved in the field trial. Their completed Certifier Profile Worksheets are shown below.

CERTIFIER PROFILE WORKSHEET

CERTIFIER NAME OR ID NUMBER	Sharon Rohde
Number of years of programming experience	5 yrs
Number of years of programming experience in asset's language	.5 yrs in C++
Education (list degrees)	MS Computer Science
Experience with Certification Tools (hours with each tool before starting certification process)	
Borland C++ IDE	10 hr
PC-Lint	3 hrs
McCabe Visual Toolset	32 hrs
C-Vision	8 hrs

CERTIFIER NAME OR ID NUMBER	Pat Aymond
Number of years of programming experience	10 yrs
Number of years of programming experience in asset's language	5 yrs
Education (list degrees)	MS, Education
Experience with Certification Tools (hours with each tool before starting certification process)	
Borland C++ IDE	2 yrs
PC-Lint	0
McCabe Visual Toolset	0
C-Vision	0

CERTIFIER NAME OR ID NUMBER	Karen Dyson
Number of years of programming experience	8
Number of years of programming experience in asset's language	.5 in Ada
Education (list degrees)	BS Civil Engineering
Experience with Certification Tools (hours with each tool before starting certification process)	
Borland C++ IDE	0 hrs
PC-Lint	0 hrs
McCabe Visual Toolset	0 hrs
C-Vision	0 hrs

Asset Description

The information contained on this worksheet is also discussed in subsection 3.2.

ASSET DESCRIPTION WORKSHEET

ASSET NAME	Labels
Origin of asset	Borland International
Application domain	Information Management
Purpose of asset	Updates and displays the contents of a mailing list.
Language	C++
Number distinct "includes" contained in the asset	5
Physical lines of code *includes blank lines and comments*	4828
Source lines of code (physical) *includes non-blank, non-comment lines*	3356 (est.)
Age of asset	1993
Version number of asset	1.0
Previous inspection and testing activities	unknown
Additional documentation	short prologue

Effort

Effort to apply the techniques for each step of the certification process was reported on the Overall Process Data Worksheet. Included in the reported effort is the effort to record defects, but not the effort learn how to use the tool. The graph in Figure 3-1 compares the actual effort to apply the techniques to the predicted, or default, effort. Default effort data is taken from CRC's Volume 3- Cost Benefit Plan.

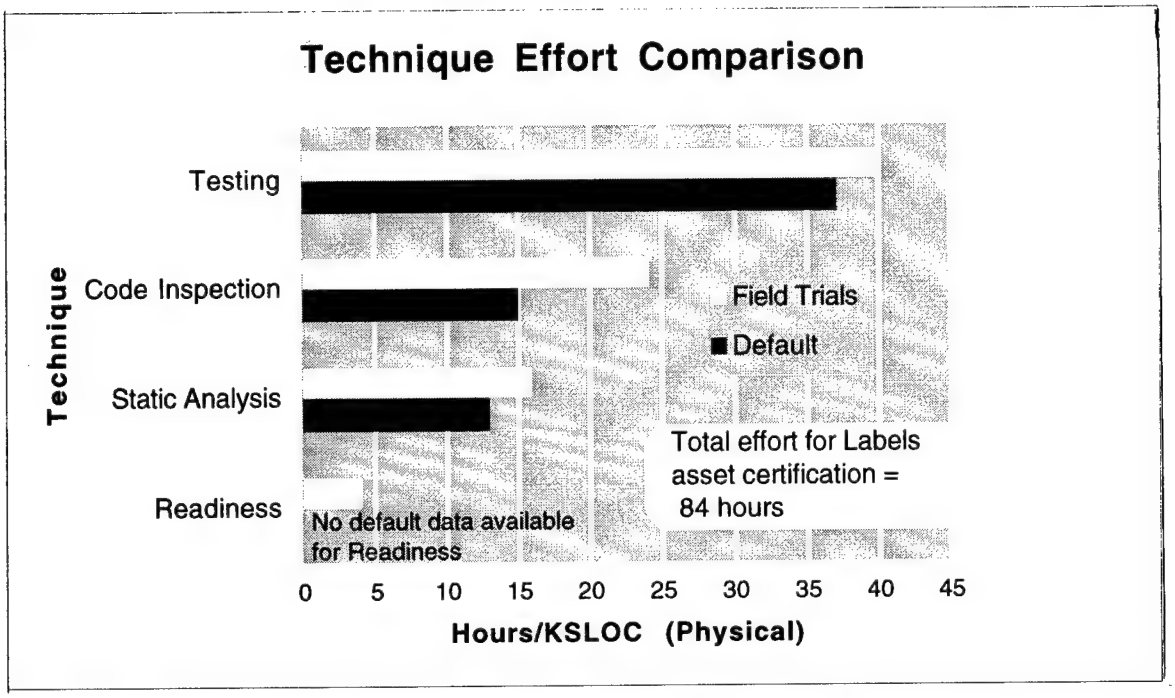

Figure 3-1. Comparison of Actual Effort to Predicted

In general, the actual effort was close to the prediction.

Since our initial effort of structural testing yielded high coverage (i.e., 97%), we elected to conclude the testing activity.

OVERALL PROCESS DATA WORKSHEET

ASSET:	Certification Step			
Labels	ASSET READINESS	STATIC ANALYSIS	CODE INSPECTION	TESTING
Certifier ID	Sharon	Sharon	Sharon	Sharon
Level of Effort (hrs)	4	16	24	40
Problems in Applying Techniques				
Problems in Using Tools	Borland required proper path settings for all included libraries and supporting reference files			Borland 5.00 and McCabe 5.2 were incompatible; upgraded to 5.01 and 5.22, respectively
Problems with Process Guidance				
Other Problems				

Defects

Many more natural defects were found in the asset during the field trial than were known prior to the start. All are recorded on defect report forms in Appendix B. Each report has an identifier that indicates the source of the report using the following codes.

Defect Report Identifier Codes

Code	Source
RD	Readiness
SA	Static Analysis
CI	Code Inspection
TE	Testing
PA	Aymond's Seeded Defect

In terms of certification, the asset passed the certification concern of Completeness, and failed in the other two concerns of Correctness and Understandability. In practice, the certifier would face the following choices:

- Reject the asset
- Report the asset as uncertified and record all known defects
- Return the asset to the donor and request repair of known defects; repeat the certification process after repairs
- Repair the defects; repeat the certification process after repairs

Some certifiers may choose to include defect repair as part of their certification process. There is some debate as to whether it would be necessary to repeat the certification process after repairs have been effected, depending on the nature and the number of the defects found. The purpose of repeating the certification would not only be to insure that the defects were repaired, but also to catch any new defects inserted as a result of the repair activity.

Counting Defects. In the following graphs and tables, unless otherwise noted, defects are counted as unique defect reports. The uniqueness criterion means that if the same defect was detected by more than one technique, it is counted only once and credited to the first technique to detect it. In filling out the defect reports, each report is limited to a single package or separately compilable file. All occurrences of the same type of error, such as a style violation, in a module are recorded on the same report, with all defective lines of code noted on the form.

Figure 3-2 shows how many defects were found by the steps in the certification process versus how many are known to exist at completion of the field trial. Defects categorized as *not found* are seeded defects.

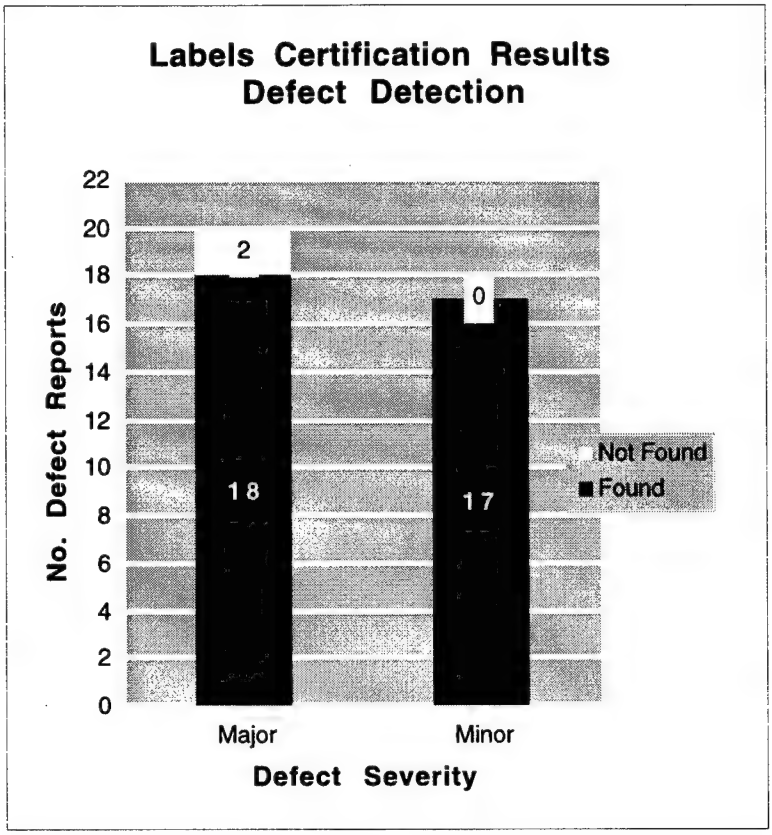

Figure 3-2. Defect Detection.

Summary of Defect Reports. The following table summarizes the defect reports logged during the certification process steps and the seeding activity. Duplicate reports are listed in the "prior step" shaded rows.

Defect Report Summary

		Defect Type					
Step	**When Found**	**Comp.**	**Data**	**I/F**	**Logic**	**Other**	**Total**
Readiness	This Step First	0	0	2	0	0	2
Static Analysis	This Step First	0	3	8	3	0	14
Code	This Step First	0	0	2	6	1	9
Inspection	Prior Step			1	0		1
Testing	This Step First	0	0	0	9	1	10
	Prior Step				0		
Seeding	Not Found	1	1	0	0	0	2
	Other Steps	0	0	4	8	0	12

Asset's Defect Profile. Figure 3-3 shows the defect profile of the asset in terms of the known defects. The defect density of the asset's major defects, including the seeded defects, is about average for C [see CRC's Cost Benefit Plan]. Major defects as we've defined them for the field trial are equivalent to what are typically reported as defects.

Defect Density

	Defect Density (defects/1000 physical lines)	
Defect Severity	**Asset's**	**Average for C**
Major	6	6
Minor	5	N/A

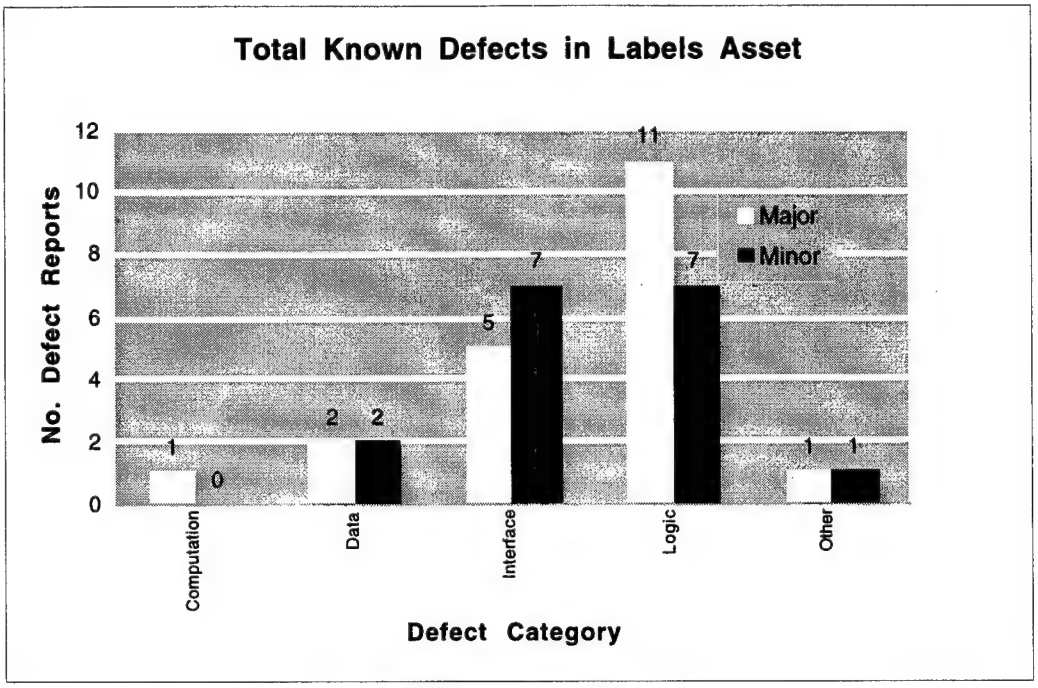

Figure 3-3. Asset's Defect Profile.

Figure 3-4 compares the asset's defect profile, including both major and minor, seeded and natural defects, to the default profile [see CRC's Cost Benefit Plan]. One notable difference is that there is a much lower proportion of computational defects. This fact could have two interpretations:

- the techniques used are not effective at finding computational defects
- the asset does not have computational defects

The second explanation is more likely, since the asset is not heavily computational in nature, only the date is computed in the labels program. No seeded defects were of the computational category. This then indicates that we cannot assess the effectiveness of the techniques at finding computational defects based on this field trial.

In certification, it will typically be the case that an individual asset's defect profile is different from the default profile of any given group of assets. The more that is known about the expected defect profile of assets to be certified, the more cost effective a process can be designed to certify them. For example, if a group of assets to be certified is known not to be computational, then you would not need to include a technique that is effective at detecting computational defects.

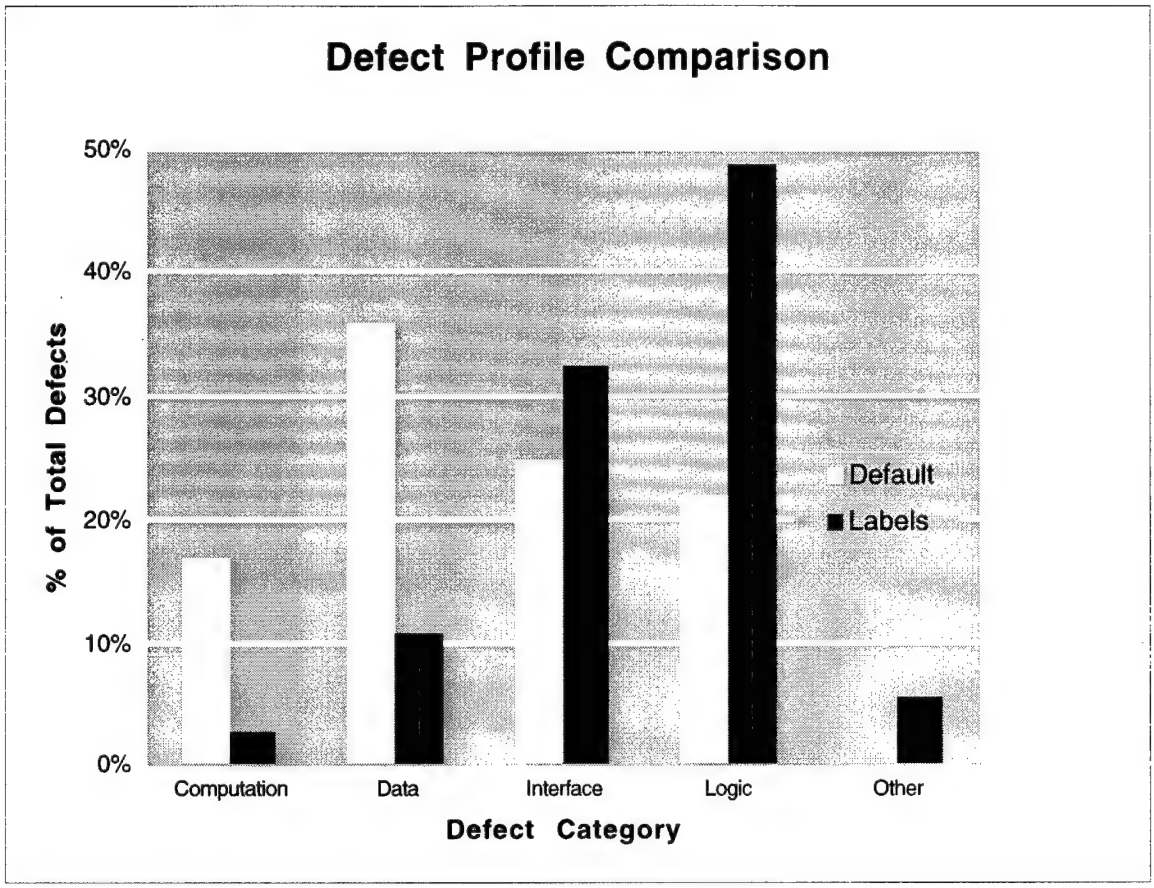

Figure 3-4. Comparison of Asset's Defect Profile to Default Profile.

Technique Effectiveness

As Figure 3-2 shows, all but two of the known major defects was found, and the two not found were seeded defects. Effectiveness of the default certification process at finding defects is better represented by the proportion of the total seeded defects found than by the proportion of known defects found. This is because there may be additional natural major defects in the asset, so the total number defects in the asset is unknown.

Effectiveness at Detecting Major Seeded Defects

Found	Known	Effectiveness
18	20	90%

Figure 3-5 shows the cumulative effectiveness of the steps in the certification process where effectiveness is defined as the proportion of known defects found. From this we can draw several important conclusions. We cannot, however, claim that the combined effectiveness of the default certification process is more than 90% because we do not know the total number of natural defects in the asset.

Furthermore, based on the effectiveness at finding seeded defects, we have reason to believe that more natural defects exist.

Readiness step. There were two major defects found during the Readiness step. Even though initially, all code needed to create an executable was available and compiled without error, we found a major defect in documentation of the code's functionality. After we upgraded our Borland compiler to operate with the upgraded McCabe Visual Toolset, we uncovered a seeded error during linking in compilation.

Static Analysis step. As Figure 3-5 shows, both major and minor defects were found by this step. The particular tool selected for this step was very good at finding defects. The 55% effectiveness rating for minor defects shown on the graph may be misleading, however. The automated tools used in this step are virtually 100% effective at finding the defects that they are *designed* to find. The effectiveness rating indicates that what the tools are designed to find were only about half of the known minor defects in the asset.

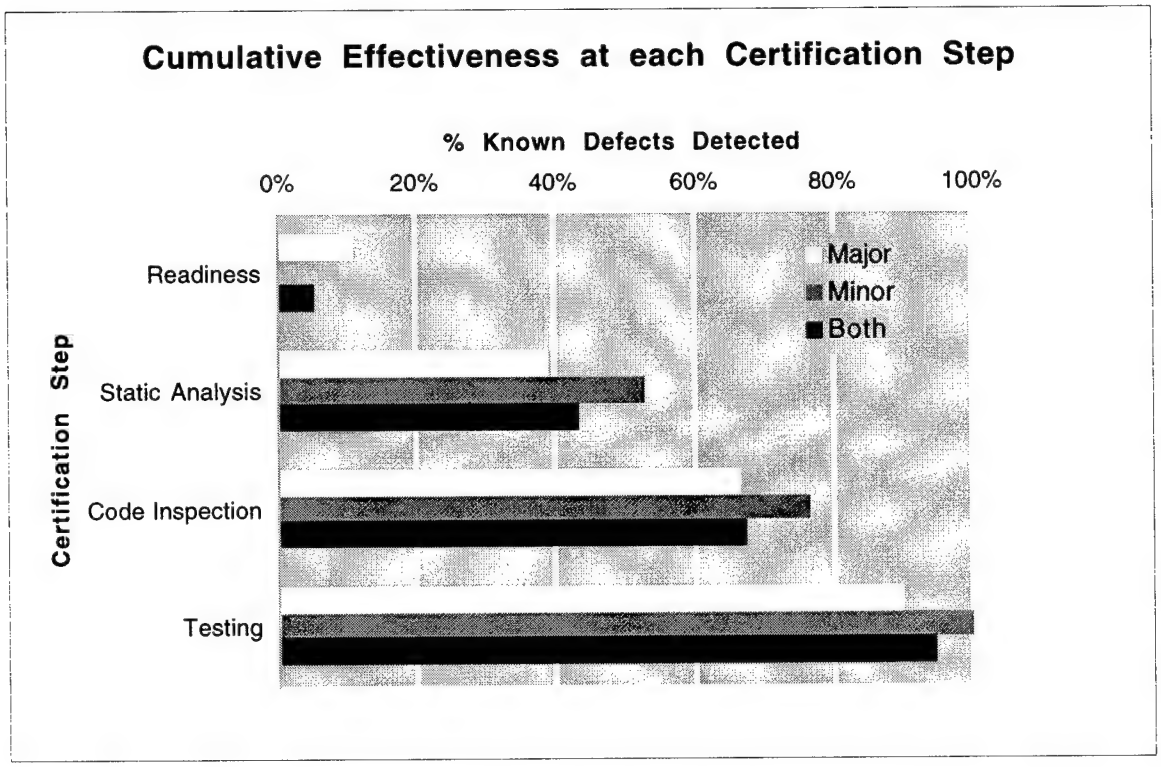

Figure 3-5. Cumulative Effectiveness of Certification Steps.

Code Inspection step. As Figure 3-5 shows, this step found about 25% of the major errors. This is lower than the industry studies that support code inspection as a useful technique to detect defects. The first field trial also had a lower than expected result. Consequently, we modified our checklist to add additional granularity to the questions in hope of improving our results. Our repeated results show that this

may not be the factor behind the shortfall. Other explanations may be the certifier skill and years of experience with the code asset language.

Testing step. Less than one-third of the defects were found in the testing step, as can be seen by subtracting the effectiveness of the code inspection step from that of the testing step in Figure 3-5. This may be low for this step, but using the cumulative effectiveness of other steps, adequate coverage was achieved.

3.4 Lessons Learned

Choice of Component Language

Even though C++ is a popular and industry-endorsed language, several flavors are in existence. These are two standard forms (i.e., ANSI/ISO and ARM), but others have created de facto standards. These varieties come into play when choosing compilers and tools that pre-process code. Different flavors of the C++ language pose interoperability problems. Some tool vendors do not support a wide variety of C++ flavors and special customizations of the tool need to be performed. These customizations are not supported by the tool vendor. These factors eventually affected the selection of the asset to be certified.

Tools that support C++ are not robust. C++ is widely acclaimed as an excellent language of choice over C, but this trend is a fairly new one. Tool vendors need additional time to provide mature tools to meet the market demand.

Defects

All defects found in the Testing Step were unique. The first field trial has some minor overlap of errors found in succeeding steps and separation was not as clearly evident as in the second field trial. Nonetheless, this finding confirms that a certification process should include a series of steps using distinct techniques designed to detect different kinds of errors. Overall, we found that each technique is special and cannot be omitted from the process.

The components used Field Trial #1 and Field Trial #2 differed in the total number of minor defects. Field Trial #1 found 77 of a total of 85 minor defects and Field Trial #2 found 17 of a total of 17. This may be due to the differences in the initial, unseeded component, as well as the differences in tools used in the two certification environments. Field Trial #1 had the advantage of AdaQuest to find minor violations of coding style whereas no such tool existed as a counterpart in the C++ certification environment. In Field Trial #2, PC-Lint was used as a thorough static analysis tool and can be thought of as a parallel tool that detects minor defects.

Many major defects were found in the earlier certification steps (i.e., prior to Code Inspection). This finding also confirms the need for a multi-step certification process. Defects found in earlier steps are less costly to find and to repair than those

found in later steps. Finding defects late in a development process (i.e., during testing) is not usually cost-effective.

Defect Categories

The categorization of defects, both seeded and natural, is difficult to assign from the definitions alone. The definitions as they appear in the CRC Code Defect Model could be improved by elaboration with additional details specific to each component language. Examples to illustrate assignments of categories would be helpful.

The Field Trial procedures would benefit by adding these examples for each kind of defect to help the Certifier and Certification Analyst to make this determination. We were able to adequately maintain consistency across the two Field Trials conducted at SPS through individual staffing.

Field Trial - Certification Tools

The configuration of the certification environment is time-consuming. We needed to artificially create the experimental environment prior to conducting the test. In a repository situation, this environment would already be established.

Installation, learning, integration, and application of tools to a particular component is very time-consuming. The activities are difficult to plan because of unknown obstacles that are encountered. It is suggested to build a three month period into the schedule for these activities alone. Using an example component that is available to the tool vendor's technical support staff is helpful in tracking bugs and errors in installation and operation of the tool.

Configuration and integration of tools is problem-fraught. Version incompatibility across tools can present problems in operation. Tools are marketed as compatible, but, as each vendor may issue monthly changes, particular versions of one tool may not work with a version of another. Upgrades to one tool may cause an new incompatibility in another tool which once functioned properly. Fortunately, for Field Trial #2, vendor support was excellent and enabled us to work through the barriers.

Since vendors issue frequent versions of their software, documentation does not match tool versions. Patches may be available, but are difficult to secure. Installation of patches may be time-consuming and problem-ridden. This presents problems with those who are learning the new tools or learning the differences in the new version.

Support for tools that instrument code is weak. For example, the instrumentation mode was not sufficiently tested using a sample program provided by the vendor with the Borland compiler and McCabe Visual Toolset. Documentation of the process was non-existent and was created "on the fly" as the problem was solved. Bugs in the tools were uncovered as the problem was resolved. We recommend

that tool vendors who have an instrumentation mode provide samples to test tool installation and functionality.

With the McCabe Battlemap, the ability to jump to the actual source line of code from the Battlemap would improve its capabilities.

Training is a requirement for high-end tools. Complex tools give sophisticated results and require a high learning curve to operate the tools properly. User documentation is typically weak; we found this to be true of both Logiscope used in the first field trial and the McCabe Visual Toolset used in the second field trial. We found that McCabe provides manuals in large binders making it difficult to find the desired information. On many occasions, once the information was found, it was incorrect and out-of-date, not matching the most current version of the tool issued. Additional expertise is required to sift through the volume of information available from the tool and interpret the results. A high level of expertise is required to learn the tools, get them up and running, use, interoperate, and interpret the results.

Training for the McCabe tools focuses on the theoretical underpinnings of the tool's complexity measures and control flow theory. We found this useful; however, another course targeting the application of the tools to a real-world situation is needed. Currently, these services are available only on an in-house consulting basis and can prove to be very costly for those on limited funds.

For complex tools, an excellent technical support staff relationship is required. The tool vendors must be responsive to tool problems, otherwise, a failure to complete could result.

Field Trial - Testing Effort

The design of the component under test greatly affects the testing effort when using a structural testing approach. The component for Field Trial #2 had a flat calling tree structure and was highly coupled across modules. Modules were small and had low control flow complexity. This structure is typical, and can be expected, for a component implemented in the C++ language. Branch coverage of 97% was easily achieved. Whereas the calling tree structure of the component in Field Trial #1 was deeper and the modules were longer and more complex, it proved difficult to achieve more than 80% branch coverage.

Certifier Skills

The suite of certification techniques that comprise the default certification process includes two techniques whose effectiveness is highly dependent upon the training and experience of the certification engineer applying the technique: code inspection and testing. These techniques are also less automated and require more human involvement than the readiness and static analysis steps. This implies that the results may not be repeatable when comparing different certification engineers. To

reduce the variability among different engineers, and to maximize the effectiveness of the techniques, training is essential.

The default process steps are intentionally ordered in terms of increasing skill level as well as increasing investment of effort, so that, for example, a failure in an early step could save wasted effort in later steps. In general, we would like the automated static analysis tools to detect as much as possible, and we view enhancements in static analysis capabilities as a valuable contribution to certification.

Effectiveness of Techniques

The combined effectiveness of all of the steps in the certification process is impressive because each step tends to find different types of defects. The second field trial confirms the results of the first, that all four steps are necessary to detect a high proportion of defects.

Figure 3-5 shows, for example, that many of the major defects would have been missed if we had only done static analysis. The Defect Report Summary table also shows that there are numerous defects that testing alone would not have found.

We recommend that defect detection be pushed to the earlier certification steps. For example, automated static analysis is a cost-effective, objective, non-cognitive technique as compared with code inspection which requires trained staff and considerable effort. The effectiveness of some techniques are contingent upon the persons using them.

The Code Inspection Step for Field Trial #1 and #2 were only moderately effective in detecting defects (i.e., 37% and 27% respectively). This may be due to a relatively small body of detectable defects over both field trials. A more definitive trial of the process would to certify multiple assets with thousands of defects. Here, in this experiment, we inserted "controlled" defects which may not necessarily be typical of the kind of defects that arise naturally.

We were impressed with the ability of the upgraded Borland compiler to detect a previously undected major error during the Readiness step. We hope that this finding is a trend among vendor upgrades as support the software developer and maintainer in detecting defects early in the software life cycle.

Modifications to the Process Guidance

General. The certification process as defined by the steps of Readiness, Static Analysis, Code Inspection, and Testing is valid. Many natural defects, as well as the seeded defects, were found in the certified COTS components. Field Trial #2 found 7 natural defects, and Field Trial #1 found 12.

Code Inspection step. In C++ with numerous, short modules, code design and its "checklist" may become more important to major and minor errors, corrections

understandability. Design appears more closely tied to implementation of function. It may be useful to add a reverse engineering tool to the certification environment to help understand code structure. We found that McCabe Visual Toolset does not provide sufficient insight.

Recommendations

Seeding defects was a difficult activity, and we cannot confirm that the defects seeded are typical of the defects that software developers and maintainers inadvertently introduce into source code. We recommend conducting a study to determine examples of defects that are typical across defect types.

Additional planned empirical research should attempt to validate the certification reuse process and procedures. Additional data could be collected for Ada, C++, components as well as other programming languages (i.e., COBOL, FORTRAN, Pascal, C, etc.) in follow-on pilot studies.

After a significant number of pilot tests, we recommend an additional phase of applying the certification reuse process to multiple components of a reuse library and collecting additional data analyses, and results for the purpose of comparison. The next phase of validation could involve multiple reuse libraries to determine the relative efficiency of those processes and procedures. The certification process could alternately be expanded to other quality concerns, other domains, and other component types.

The disappointing results achieved in the Code Inspection step, suggest a topic for future research, i.e., the study of ways to make code inspection more effective. This research topic is also of interest to software maintainers who routinely struggle with the comprehension of code written by others.

The results of the Field Trials is of interest to the software/systems community. The technical paper and presentation of the first field trial at the IEEE International Conference on Engineering of Complex Computer Systems '96 (ICECCS) was well-received and drew additional conversation from its participants. We intend to follow-up with an additional paper about the second field trial and its comparison to the first at a future conference.

Acronyms

CRC	Certification of Reusable Software Components
DD	Decision-to-Decision
FTR	Final Technical Report

Appendix A: Data Collection Forms

ASSET DESCRIPTION WORKSHEET

ASSET NAME	
Origin of asset	
Application domain	
Purpose of asset	
Language	
Number distinct "includes" contained in the asset	
Physical lines of code (non-blank lines)	
Lines of some code	
Age of asset	
Version number of asset	
Previous inspection and testing activities	
Additional documentation	

CERTIFIER PROFILE WORKSHEET

CERTIFIER NAME OR ID NUMBER	
Number of years of programming experience	
Number of years of programming experience in asset's language	
Education (list degrees)	
Experience with Certification Tools (hours with each tool) (note: before starting certification process)	____ Borland C++ IDE, PC Windows 95 ____ PC-Lint _____ C-Vision _____ McCabe Visual Toolset

◆ Certification Defect Report ◆

Defect Report Identifier _____

Severity ☐ Major ☐ Minor

Asset Identifier _____

Originator _____

Defect Type
- ☐ Computational
- ☐ Data
- ☐ Interface
- ☐ Logic
- ☐ Other _____

Technique Used
- ☐ Readiness
- ☐ Static Analysis
  - ☐ Data Flow
  - ☐ Order Dependency
  - ☐ Alias Usage
  - ☐ Unreachable Code
  - ☐ Style Guideline
  - ☐ Other _____
- ☐ Code Inspection ☐ Item _____
- ☐ Testing
  - ☐ Functional Test Case
  - ☐ Structural Test Case
  - ☐ Other _____

Tool Used
- ☐ C-Vision
- ☐ PC-Lint
- ☐ McCabe Toolset
- ☐ C++ env. (compiler, etc.)
- ☐ None

Code Line Number(s) _____

Effort to Isolate _____

Effort to Repair _____

Description of Defect:

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

OVERALL PROCESS DATA WORKSHEET

ASSET:	ASSET READINESS	STATIC ANALYSIS	CODE INSPECTION	TESTING
Certifier ID				
Level of Effort (hrs)				
Problems in Applying Techniques				
Problems in Using Tools				
Problems with Process Guidance				
Other Problems				

Appendix B: Certification Defect Reports

■ Certification Defect Report ■

Defect Report Identifier RD_001

Severity Major

Unit Name date_.h

Originator Sharon

Defect Category Interface

Tool Used Borland C++ 5.01 IDE

Certification Step 1-Readiness

Specific Technique Other

Code Line Numbers 253, 256

Effort to Isolate

Effort to Repair

Description of Defect:

Fatal error during link with new version of compiler identifed problems on these lines.

Removed PA_004 to correct.

Certification Concern Correctness

Defect Source Seeded

Related Report removed PA_004

■ Certification Defect Report ■

Defect Report Identifier RD_002

Severity Minor

Unit Name objstrm_.h

Originator Sharon

Defect Category Computational

Tool Used Borland C++ 5.01 IDE

Certification Step 1-Readiness

Specific Technique

Code Line Numbers 626

Effort to Isolate

Effort to Repair

Description of Defect:

Compiler warning: conversion may lose significant digits. (Warning was not present with Borland 5.00 compiler.) Not able to discern whether this is a defect or intentional.

Note: this inline function was never executed--unable to create a test case to exercise that branch.

Certification Concern Correctness

Defect Source Seeded

Related Report partial PA_010

■ Certification Defect Report ■

Defect Report Identifier RD_003

Severity Major

Unit Name date_.h

Originator Sharon

Defect Category Interface

Tool Used Borland C++ 5.01 IDE

Certification Step 1-Readiness

Specific Technique

Code Line Numbers Compiler indicated line 255 which used defective definition from line 106

Effort to Isolate

Effort to Repair

Description of Defect:

Compiler error: "TDate::operator -=(int)' is not a member of 'TDate'". (Warning not present with Borland 5.00 compiler)

Found to be due to line 106. Removed PA_003 by changing from "==" to "-=" and "dt" to "dd".

Certification Concern Correctness

Defect Source Seeded

Related Report found & removed PA_003

■ Certification Defect Report ■

Defect Report Identifier SA_001 **Severity** Major

Unit Name cstring.h

Originator Sharon

Defect Category Logic

Tool Used PC-Lint

Certification Step 2-Static Analysis

Specific Technique Error Checking

Code Line Numbers 133, 139, 506

Effort to Isolate

Effort to Repair

Description of Defect:

Syntax Error 10: Expecting identifier.
"{" on a line by itself

Certification Concern Correctness

Defect Source Natural

Related Report

■ Certification Defect Report ■

Defect Report Identifier SA_002

Severity Major

Unit Name cstring.h

Originator Sharon

Defect Category Data

Tool Used PC-Lint

Certification Step 2-Static Analysis

Specific Technique Error Checking

Code Line Numbers 148, 150, 152, 154, 156, 158, 159, 161, 162, 164, 165, 168, 172, 174, 195, 206, 214, 217, 228, 286, 290, 297, 308, 347, 349, 380, 382, 490, 556, 558, 559, 560, 564

Effort to Isolate

Effort to Repair

Description of Defect:

Syntax Error 49: Expected a type. "xalloc"

Certification Concern Correctness

Defect Source Natural

Related Report

■ Certification Defect Report ■

Defect Report Identifier SA_003

Severity Major

Unit Name cstring.h

Originator Sharon

Defect Category Logic

Tool Used PC-Lint

Certification Step 2-Static Analysis

Specific Technique Error Checking

Code Line Numbers 654, 659

Effort to Isolate

Effort to Repair

Description of Defect:

Syntax Error 10: Expecting identifier. "xmsg"

Certification Concern Correctness

Defect Source Natural

Related Report

■ Certification Defect Report ■

Defect Report Identifier SA_004 **Severity** Minor

Unit Name labels_.cpp

Originator Sharon

Defect Category Interface

Tool Used PC-Lint

Certification Step 2-Static Analysis

Specific Technique Programming Standards

Code Line Numbers 35

Effort to Isolate

Effort to Repair

Description of Defect:

Warning 537: Repeated include file: 'c:\lint\fstream.h'

Certification Concern Understandability

Defect Source Natural

Related Report

■ Certification Defect Report ■

Defect Report Identifier SA_005

Severity Minor

Unit Name labels_.cpp

Originator Sharon

Defect Category Logic

Tool Used PC-Lint

Certification Step 2-Static Analysis

Specific Technique Programming Standards

Code Line Numbers 151, 153, 165, 170, 305, 306, 313, 317, 405, 410, 411, 415, 421, 428, 432, 439, 503, 506, 509, 512, 515, 518, 534, 537

Effort to Isolate

Effort to Repair

Description of Defect:

Warning 534: Ignoring return value of operators.

Certification Concern Correctness

Defect Source Natural

Related Report

■ Certification Defect Report ■

Defect Report Identifier SA_006 **Severity** Minor

Unit Name labels_.cpp

Originator Sharon

Defect Category Data

Tool Used PC-Lint

Certification Step 2-Static Analysis

Specific Technique Type Checking

Code Line Numbers 167, 315, 423

Effort to Isolate

Effort to Repair

Description of Defect:

Warning 641: Converting enum to int.
"in.clear(ios::failbit);"

Certification Concern Understandability

Defect Source Natural

Related Report

■ Certification Defect Report ■

Defect Report Identifier SA_007 **Severity** Minor

Unit Name labels_.cpp

Originator Sharon

Defect Category Interface

Tool Used PC-Lint

Certification Step 2-Static Analysis

Specific Technique Error Checking

Code Line Numbers 294

Effort to Isolate

Effort to Repair

Description of Defect:

Info 1702: operator 'operator<' is both an ordinary function 'operator<(const TPWObj &, const TPWObj &)' and a member function 'TDate::operator<(const TDate &) const'

Certification Concern Correctness

Defect Source Natural

Related Report

■ Certification Defect Report ■

Defect Report Identifier SA_008

Severity Minor

Unit Name labels_.cpp

Originator Sharon

Defect Category Interface

Tool Used PC-Lint

Certification Step 2-Static Analysis

Specific Technique Error Checking

Code Line Numbers 374, 479

Effort to Isolate

Effort to Repair

Description of Defect:

Info 1712: default constructor not defined for classes 'TSubscriptionList' and 'TNewSubscribers'

Certification Concern Completeness

Defect Source Natural

Related Report

■ Certification Defect Report ■

Defect Report Identifier SA_009 **Severity** Minor

Unit Name labels_.cpp

Originator Sharon

Defect Category Interface

Tool Used PC-Lint

Certification Step 2-Static Analysis

Specific Technique

Code Line Numbers 382

Effort to Isolate

Effort to Repair

Description of Defect:

Warning 1541: member TSubscriptionList::Subscriptions (line 369) possibly not initialized by constructor

Certification Concern Completeness

Defect Source Natural

Related Report

■ Certification Defect Report ■

Defect Report Identifier SA_010 **Severity** Minor

Unit Name labels_.cpp

Originator Sharon

Defect Category Data

Tool Used PC-Lint

Certification Step 2-Static Analysis

Specific Technique Error Checking

Code Line Numbers 477

Effort to Isolate

Effort to Repair

Description of Defect:

Info 1725: class member "TNewSubscribers::List" is a reference

Certification Concern Correctness

Defect Source Natural

Related Report

■ Certification Defect Report ■

Defect Report Identifier SA_011 **Severity** Major

Unit Name labels_.cpp

Originator Sharon

Defect Category Interface

Tool Used PC-Lint

Certification Step 2-Static Analysis

Specific Technique Error Checking

Code Line Numbers 504, 507, 510, 513, 516

Effort to Isolate

Effort to Repair

Description of Defect:

C++ Syntax Error 1036: ambiguous reference to constructor; candidates:'string::string(const char*) and string::string(const char far*). Assignment statements.

Certification Concern Correctness

Defect Source Natural

Related Report

■ Certification Defect Report ■

Defect Report Identifier SA_012 **Severity** Major

Unit Name labels_.cpp

Originator Sharon

Defect Category Interface

Tool Used PC-Lint

Certification Step 2-Static Analysis

Specific Technique Error Checking

Code Line Numbers 611, 614

Effort to Isolate

Effort to Repair

Description of Defect:

C++ Syntax Error 1036: ambiguous reference to constructor; candidates: `string::string(const char *) and string::string(const char far *). Reference to an array of arguments.

Certification Concern Correctness

Defect Source Natural

Related Report

■ Certification Defect Report ■

Defect Report Identifier SA_013 **Severity** Minor

Unit Name labels_.cpp

Originator Sharon

Defect Category Interface

Tool Used PC-Lint

Certification Step 2-Static Analysis

Specific Technique Error Checking

Code Line Numbers 174, 179, 184, 189

Effort to Isolate

Effort to Repair

Description of Defect:

Info 1714: Member functions not referenced:
TSubscriber::GetName(void) const,
TSubscriber::GetAddress(void) const,
TSubscriber::GetCity(void) const, and
TSubscriber::GetState(void) const

Certification Concern Understandability

Defect Source Natural

Related Report

■ Certification Defect Report ■

Defect Report Identifier SA_014

Severity Minor

Unit Name labels_.cpp

Originator Sharon

Defect Category Interface

Tool Used PC-Lint

Certification Step 2-Static Analysis

Specific Technique Error Checking

Code Line Numbers 282, 287

Effort to Isolate

Effort to Repair

Description of Defect:

Info 1714: Member functions not referenced:
TSubscriptionInfo::operator==(const TSubscriptionInfo &) and
TSubscriptionInfo::operator<(const TSubscriptionInfo &)

Certification Concern Understandability

Defect Source Natural

Related Report

■ Certification Defect Report ■

Defect Report Identifier SA_015

Severity Minor

Unit Name labels_.cpp

Originator Sharon

Defect Category Interface

Tool Used PC-Lint

Certification Step 2-Static Analysis

Specific Technique Error Checking

Code Line Numbers 408

Effort to Isolate

Effort to Repair

Description of Defect:

Info 1714: Member function not referenced:
TSubscriptionList::WriteStream(opstream &)

Note: if checked, this warning might have lead to discovery of seeded error PA_001

Certification Concern Understandability

Defect Source Seeded

Related Report related to PA_001

■ Certification Defect Report ■

Defect Report Identifier CI_001

Severity Major

Unit Name date_.h

Originator Sharon

Defect Category Interface

Tool Used CI Checklist

Certification Step 3-Code Inspection

Specific Technique Code Inspection Item C.14.C

Code Line Numbers 256

Effort to Isolate

Effort to Repair

Description of Defect:

Incorrect assigment using compound operator

Certification Concern Correctness

Defect Source Seeded

Related Report found PA_004

■ Certification Defect Report ■

Defect Report Identifier CI_002

Severity Major

Unit Name date_.h

Originator Sharon

Defect Category Logic

Tool Used CI Checklist

Certification Step 3-Code Inspection

Specific Technique Code Inspection Item D.10.C

Code Line Numbers 272

Effort to Isolate

Effort to Repair

Description of Defect:

Lower and upper bounds of range incorrectly assigned

Certification Concern Correctness

Defect Source Seeded

Related Report found PA_005

■ Certification Defect Report ■

Defect Report Identifier CI_003

Severity Minor

Unit Name labels_.cpp

Originator Sharon

Defect Category Interface

Tool Used CI Checklist

Certification Step 3-Code Inspection

Specific Technique Code Inspection Item D.10.C

Code Line Numbers 605

Effort to Isolate

Effort to Repair

Description of Defect:

Type assignment for return of main is inconsistent with comments.

Certification Concern Correctness

Defect Source Seeded

Related Report found PA_014

■ Certification Defect Report ■

Defect Report Identifier CI_004 **Severity** Major

Unit Name labels_.cpp

Originator Sharon

Defect Category Interface

Tool Used CI Checklist

Certification Step 3-Code Inspection

Specific Technique Code Inspection Item I.24.C

Code Line Numbers 386

Effort to Isolate

Effort to Repair

Description of Defect:

In-type call for an out-type operation

Certification Concern Correctness

Defect Source Seeded

Related Report found PA_001

■ Certification Defect Report ■

Defect Report Identifier CI_005 **Severity** Major

Unit Name labels_.cpp

Originator Sharon

Defect Category Interface

Tool Used CI Checklist

Certification Step 3-Code Inspection

Specific Technique Code Inspection Item I.24.C

Code Line Numbers 387

Effort to Isolate

Effort to Repair

Description of Defect:

Read operation applied to an out-type parameter

Certification Concern Correctness

Defect Source Seeded

Related Report found PA_001

■ Certification Defect Report ■

Defect Report Identifier CI_006

Severity Major

Unit Name labels_.cpp

Originator Sharon

Defect Category Logic

Tool Used CI Checklist

Certification Step 3-Code Inspection

Specific Technique Code Inspection Item L.01.C

Code Line Numbers 414

Effort to Isolate

Effort to Repair

Description of Defect:

Incorrect assignment of compound boolean expression for while loop condition

Certification Concern Correctness

Defect Source Seeded

Related Report found PA_012

■ Certification Defect Report ■

Defect Report Identifier CI_007

Severity Major

Unit Name listimp_.h

Originator Sharon

Defect Category Logic

Tool Used CI Checklist

Certification Step 3-Code Inspection

Specific Technique Code Inspection Item L.05.C

Code Line Numbers 83

Effort to Isolate

Effort to Repair

Description of Defect:

Incorrect boolean expression for if statement

Certification Concern Correctness

Defect Source Seeded

Related Report partial PA_006

■ Certification Defect Report ■

Defect Report Identifier CI_008 **Severity** Minor

Unit Name labels_.cpp

Originator Sharon

Defect Category Logic

Tool Used CI Checklist

Certification Step 3-Code Inspection

Specific Technique Code Inspection Item L.26.U

Code Line Numbers 414-415

Effort to Isolate

Effort to Repair

Description of Defect:

Braces missing from while loop with one statement

Certification Concern Understandability

Defect Source Natural

Related Report

■ Certification Defect Report ■

Defect Report Identifier CI_009

Severity Minor

Unit Name listimp_.h

Originator Sharon

Defect Category Logic

Tool Used CI Checklist

Certification Step 3-Code Inspection

Specific Technique Code Inspection Item L.26.U

Code Line Numbers 299-300, 361-365, 580-581, 705-706, 731-735, 852-855, 1011-1013

Effort to Isolate

Effort to Repair

Description of Defect:

Braces missing from while loop with one statement or with if-else block.

Certification Concern Understandability

Defect Source Natural

Related Report

■ Certification Defect Report ■

Defect Report Identifier CI_010

Severity Minor

Unit Name objstrm_.h

Originator Sharon

Defect Category Logic

Tool Used CI Checklist

Certification Step 3-Code Inspection

Specific Technique Code Inspection Item L.28.C

Code Line Numbers 648-650, 756-758, 760-763, 765-767, 769-771, 786-788, 790-793, 795-798

Effort to Isolate

Effort to Repair

Description of Defect:

Missing return in inline function call

Certification Concern Understandability

Defect Source Natural

Related Report

■ Certification Defect Report ■

Defect Report Identifier CI_011

Severity Major

Unit Name labels_.cpp

Originator Sharon

Defect Category Logic

Tool Used CI Checklist

Certification Step 3-Code Inspection

Specific Technique Code Inspection Item L.29.C

Code Line Numbers 429

Effort to Isolate

Effort to Repair

Description of Defect:

For loop ounter not initialized to zero

Certification Concern Correctness

Defect Source Seeded

Related Report found PA_013

■ Certification Defect Report ■

Defect Report Identifier CI_012

Severity Major

Unit Name labels_.cpp

Originator Pat

Defect Category Other

Tool Used CI Checklist

Certification Step 3-Code Inspection

Specific Technique Code Inspection Item O.01.U

Code Line Numbers 1-29

Effort to Isolate

Effort to Repair

Description of Defect:

Description of what the component does and how it does it was lacking in detail

Certification Concern Understandability

Defect Source Natural

Related Report

■ Certification Defect Report ■

Defect Report Identifier TE_001

Severity Major

Unit Name objstrm_.h

Originator Sharon

Defect Category Logic

Tool Used McCabe Toolset 5.2

Certification Step 4-Testing

Specific Technique Other

Code Line Numbers 1020

Effort to Isolate

Effort to Repair

Description of Defect:

Syntax error on this line detected by pre-processor for instrumentation of code.

Removed PA_011 to correct this.

Certification Concern Correctness

Defect Source Seeded

Related Report removed PA_011

■ Certification Defect Report ■

Defect Report Identifier TE_002

Severity Major

Unit Name objstrm_.h

Originator Sharon

Defect Category Logic

Tool Used Borland C++ 5.01 IDE

Certification Step 4-Testing

Specific Technique

Code Line Numbers 299

Effort to Isolate 0.5 hr

Effort to Repair

Description of Defect:

Program aborted using test case supplied with component. Traced to line 299 in debugger.

Removed PA_009 to correct this.

Certification Concern Correctness

Defect Source Seeded

Related Report removed PA_009

■ Certification Defect Report ■

Defect Report Identifier TE_003

Severity Major

Unit Name listimp_.h

Originator Sharon

Defect Category Logic

Tool Used Borland C++ 5.01 IDE

Certification Step 4-Testing

Specific Technique Other

Code Line Numbers 719

Effort to Isolate 0.25 hr

Effort to Repair

Description of Defect:

Program aborted on test case supplied with component. Traced to this line with debugger.

Removed PA_007 to correct this.

Certification Concern Correctness

Defect Source Seeded

Related Report removed PA_007

■ Certification Defect Report ■

Defect Report Identifier TE_004

Severity Major

Unit Name listimp_.h

Originator Sharon

Defect Category

Tool Used Borland C++ 5.01 IDE

Certification Step 4-Testing

Specific Technique Functional Test Case test_01 with listimp_.h

Code Line Numbers

Effort to Isolate

Effort to Repair

Description of Defect:

Program aborts with numerous test cases (test_01, test_05, test_06, and test_11).

Reverted to original version of listimp.h, removing defects PA_006 and PA_008 to correct this.

Certification Concern Correctness

Defect Source Seeded

Related Report removed PA_006 and PA_008

■ Certification Defect Report ■

Defect Report Identifier TE_005

Severity Minor

Unit Name labels_.cpp

Originator Sharon

Defect Category Logic

Tool Used None

Certification Step 4-Testing

Specific Technique Functional Test Case test_02 with listimp.h

Code Line Numbers

Effort to Isolate

Effort to Repair

Description of Defect:

Test case with invalid input file--new subscription file is a master file format. Program does not diagnose the problem.

Certification Concern Correctness

Defect Source Natural

Related Report

■ Certification Defect Report ■

Defect Report Identifier TE_006

Severity Minor

Unit Name labels_.cpp

Originator Sharon

Defect Category Logic

Tool Used None

Certification Step 4-Testing

Specific Technique Functional Test Case test_04 with listimp.h

Code Line Numbers

Effort to Isolate

Effort to Repair

Description of Defect:

Test case with very long address string. Program reports subscription is invalid, but does not report which part of subscription is incorrect. Documentation does not indicate a limit for subscription field length.

Certification Concern Understandability

Defect Source Natural

Related Report

■ Certification Defect Report ■

Defect Report Identifier TE_007

Severity Minor

Unit Name labels_.cpp

Originator Sharon

Defect Category Other

Tool Used None

Certification Step 4-Testing

Specific Technique Functional Test Case test_05 with listimp.h

Code Line Numbers

Effort to Isolate

Effort to Repair

Description of Defect:

Test case contains state as a word instead of a two-letter abbreviation. Program accepts this. Documentation does not indicate how the state should be input--should be more specific.

Certification Concern Understandability

Defect Source Natural

Related Report

■ Certification Defect Report ■

Defect Report Identifier TE_008

Severity Minor

Unit Name labels_.cpp

Originator Sharon

Defect Category Logic

Tool Used None

Certification Step 4-Testing

Specific Technique Functional Test Case test_06 with listimp.h

Code Line Numbers

Effort to Isolate

Effort to Repair

Description of Defect:

Test case with incorrect zip code "000000". Program accepts this; does not check validity of zip code. Documentation does not describe required zip code format.

Certification Concern Understandability

Defect Source Natural

Related Report

■ Certification Defect Report ■

Defect Report Identifier TE_009

Severity Major

Unit Name labels_.cpp

Originator Sharon

Defect Category Logic

Tool Used None

Certification Step 4-Testing

Specific Technique Functional Test Case test_09 with listimp.h

Code Line Numbers

Effort to Isolate

Effort to Repair

Description of Defect:

Test case specifying non-existent input files on command line. Program does not give any indication of an error.

Certification Concern Understandability

Defect Source Natural

Related Report

■ Certification Defect Report ■

Defect Report Identifier PA_001 **Severity** Major

Unit Name labels.cpp

Originator Pat

Defect Category Interface

Tool Used

Certification Step

Specific Technique

Code Line Numbers 386, 387

Effort to Isolate

Effort to Repair

Description of Defect:

Changed write to read on output file, so it no longer writes subscription list to master file when subscription list object is destroyed.

386 changed "ofpstream" to "ifpstream"
387 changed "writestream" to "readstream"

Certification Concern Correctness

Defect Source Seeded

Related Reports CI_004, CI_005, SA_015

■ Certification Defect Report ■

Defect Report Identifier PA_002 **Severity** Major

Unit Name date.h

Originator Pat

Defect Category Data

Tool Used

Certification Step

Specific Technique

Code Line Numbers 31

Effort to Isolate

Effort to Repair

Description of Defect:

Changed value of constant for Julian date of 1/1/1901 from "2415386L" to "1415386L".

Certification Concern Correctness

Defect Source Seeded

Related Reports

■ Certification Defect Report ■

Defect Report Identifier PA_003 **Severity** Major

Unit Name date.h

Originator Pat

Defect Category Interface

Tool Used

Certification Step

Specific Technique

Code Line Numbers 106

Effort to Isolate

Effort to Repair

Description of Defect:

Changed data type from integer "dd" to constant "dt", plus changed operator from -+ to ==.

Certification Concern Correctness

Defect Source Seeded

Related Reports PA_004, removed RD_003

■ Certification Defect Report ■

Defect Report Identifier PA_004 **Severity** Major

Unit Name date.h

Originator Pat

Defect Category Interface

Tool Used

Certification Step

Specific Technique

Code Line Numbers 253, 256

Effort to Isolate

Effort to Repair

Description of Defect:

Changed inline function from assignment "-=" to equality "==".

Certification Concern Correctness

Defect Source Seeded

Related Reports CI_001, removed RD_001

■ Certification Defect Report ■

Defect Report Identifier PA_005 **Severity** Major

Unit Name date.h

Originator Pat

Defect Category Logic

Tool Used

Certification Step

Specific Technique

Code Line Numbers 272

Effort to Isolate

Effort to Repair

Description of Defect:

Changed inline function AssertIndexOfMonth so that months of Jan & Dec would not be valid. Changed "m >= 1 && m <= 12" to "m >=2 && m <= 11".

(Note: this function never executed.)

Certification Concern Correctness

Defect Source Seeded

Related Reports CI_002

■ Certification Defect Report ■

Defect Report Identifier PA_006

Severity Major

Unit Name listimp.h

Originator Pat

Defect Category Logic

Tool Used

Certification Step

Specific Technique

Code Line Numbers 82, 83, 91

Effort to Isolate

Effort to Repair

Description of Defect:

Instead of zeroing out the list element counter, it was set to 1 in the TMListBlockInitializer destructor.

Certification Concern Correctness

Defect Source Seeded

Related Reports removed TE_004, CI_007 is partial

■ Certification Defect Report ■

Defect Report Identifier PA_007

Severity Major

Unit Name listimp.h

Originator Pat

Defect Category Logic

Tool Used

Certification Step

Specific Technique

Code Line Numbers 719

Effort to Isolate

Effort to Repair

Description of Defect:

In ForEach list iterator while loop should continue while checking for inequality; this has been changed to check for equality.
Changed "cur->Next != cur" to "cur->Next == cur".

Certification Concern Correctness

Defect Source Seeded

Related Reports removed TE_003

■ Certification Defect Report ■

Defect Report Identifier PA_008

Severity Major

Unit Name listimp.h

Originator Pat

Defect Category Logic

Tool Used

Certification Step

Specific Technique

Code Line Numbers 889

Effort to Isolate

Effort to Repair

Description of Defect:

Evaluates a list of pointers to objects of type T. Defect changes notation from class name to arithmetic operators. Changed "<T" to ">T".

Certification Concern Correctness

Defect Source Seeded

Related Reports removed TE_004

■ Certification Defect Report ■

Defect Report Identifier PA_009

Severity Major

Unit Name objstrm.h

Originator Pat

Defect Category Logic

Tool Used

Certification Step

Specific Technique

Code Line Numbers 299

Effort to Isolate

Effort to Repair

Description of Defect:

Change syntax of line so that address of object is not stored in the database. Changed "*>" to ">".

Certification Concern Correctness

Defect Source Seeded

Related Reports removed TE_002

■ Certification Defect Report ■

Defect Report Identifier PA_010 **Severity** Major

Unit Name objstrm.h

Originator Pat

Defect Category Computational

Tool Used

Certification Step

Specific Technique

Code Line Numbers 626

Effort to Isolate

Effort to Repair

Description of Defect:

Changed return value computation of inline function clear by changing from "hardfail" to "basefield", where "basefield" is a different data type (long int vs int).

Certification Concern Correctness

Defect Source Seeded

Related Reports

■ Certification Defect Report ■

Defect Report Identifier PA_011 **Severity** Major

Unit Name objstrm.h

Originator Pat

Defect Category Logic

Tool Used

Certification Step

Specific Technique

Code Line Numbers 1020

Effort to Isolate

Effort to Repair

Description of Defect:

Improper terminator in switch statement. Changed from "break" to "switch". in first case.

Certification Concern Correctness

Defect Source Seeded

Related Reports removed TE_001

■ Certification Defect Report ■

Defect Report Identifier PA_012 **Severity** Major

Unit Name labels.cpp

Originator Pat

Defect Category Logic

Tool Used

Certification Step

Specific Technique

Code Line Numbers 414

Effort to Isolate

Effort to Repair

Description of Defect:

Wrong condition in while loop. Changed "iter != 0" to "iter == 7". Will not correctly write contents of subscription list to output file.

Certification Concern Correctness

Defect Source Seeded

Related Reports CI_006

■ Certification Defect Report ■

Defect Report Identifier PA_013 **Severity** Major

Unit Name labels.cpp

Originator Pat

Defect Category Logic

Tool Used

Certification Step

Specific Technique

Code Line Numbers 429

Effort to Isolate

Effort to Repair

Description of Defect:

Incorrect initialization of for loop iterator. Changed "i=0" to "i=11". Will not read subscriptions from master file unless count > 11.

Certification Concern Correctness

Defect Source Seeded

Related Reports CI_011

■ Certification Defect Report ■

Defect Report Identifier PA_014 **Severity** Minor

Unit Name labels.cpp

Originator Pat

Defect Category Interface

Tool Used

Certification Step

Specific Technique

Code Line Numbers 605

Effort to Isolate

Effort to Repair

Description of Defect:

Type declaration of main return does not match commented description. Changed "int" to "unsigned int".

Certification Concern Correctness

Defect Source Seeded

Related Reports CI_014

*U.S. GOVERNMENT PRINTING OFFICE: 1997-509-127-61068

MISSION OF ROME LABORATORY

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

a. Conducts vigorous research, development and test programs in all applicable technologies;

b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;

c. Provides a full range of technical support to Air Force Material Command product centers and other Air Force organizations;

d. Promotes transfer of technology to the private sector;

e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.